Package Templates: Design, Experimentation and Implementation

Evolving a Mechanism for Reuse and Adaption of Class Collections

Eyvind W. Axelsen

December 2013

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1485

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

Abstract

This thesis presents a novel approach to reuse and adaption of collections of classes in object-oriented languages. The approach is called *Package Templates*, and is based on ideas first published by Krogdahl in 2001.

Package Templates presents the programmer with a mechanism for writing modules, in the form of collections of classes, in a way that that allows these modules to be adapted to the problem at hand before they are used in a program. The modules can contain class hierarchies, and adaptions can be applied at any level of such a hierarchy. All adaptions are applied when the module is instantiated, and can include renaming, class merging, high-level parameterization, refinement and retroactive interface implementations.

The thesis presents the mechanism in the context of both statically and dynamically typed programming languages, and explores and discusses the design space for both variants, and their respective tradeoffs in terms of flexibility, expressive power and safety.

Prototype implementations, case studies and example libraries are presented in order to both demonstrate and assess the utility of the mechanism.

Acknowledgements

This work has been done in the context of the SWAT project (Semantics-preserving Weaving — Advancing the Technology) at the Department of Informatics, University of Oslo, and has been funded by The Research Council of Norway under grant number 167172/V30.

I am very grateful for the guidance, assistance, and inspiration provided by my main supervisor Stein Krogdahl. He has been, and is, a constant source of encouragement and enlightenment, and without his thorough feedback and endless positivity this work would not have come to its completion. This thesis is also based on initial work performed by Krogdahl, who early on understood that adaption and generalization on the level of modules were important features that could be very useful in many object-oriented programming languages.

Furthermore, I would like to thank my two co-supervisors. Birger Møller-Pedersen has been an important sparring partner for a large amount of the topics presented herein, and is also a co-author on two of the papers. I am grateful for him sharing his insights on many topics with me, and for his positive attitude towards this PhD project. Øystein Haugen has participated in several discussions pertaining to the subject of this thesis and related fields, and has provided important feedback in particular for talks that were to be held at various conferences.

My fellow students in the SWAT project and related projects, Fredrik Sørensen, Roy Grønmo, Jon Oldevik, Henning Berg, and Weiqing Zhang, have all been part of making my time at the University of Oslo both fun and interesting, and we have enjoyed many good discussions pertaining to topics both inside and well outside the realm of computer science. Sørensen, in particular, has been important for this thesis, and has provided several ideas and insights that are presented here. He is a co-author of two of the papers herein.

Several master students have been involved in the development of the PT compiler. In particular, I would like to thank Eivind Gard Lund, Ivar Refsdal, Steinar Kaldager, Daniel Rødskog, Lars Jørgen Solberg and Kenneth Solbø Andersen for dedicating their time and programming skills to this project.

I am grateful to the Department of Informatics, and the University of Oslo, for providing me with a good working environment. There are many smart and friendly people employed there, who contribute to making it an inspirational and enjoyable place to work and study. I am also grateful to the two companies that have employed me outside of the university during my work with this thesis: Compugroup Medical Norway and Fürst Medical Laboratory. They have both provided me with the freedom

and flexibility needed to complete this thesis.

I thank my mother Mari for her everlasting support, and for always believing in me, no matter what endeavor I undertake. My father Bjørn, who passed away much too soon, instilled in me from an early age a strong sense of curiosity and a desire for knowledge, which I believe has been a deciding factor in leading me down this path; for this I am grateful.

Last, but not least: as important as reuse and adaption of software modules is, there are indeed things in life that I treasure even more. I would like to extend my sincerest gratitude to my lovely wife Fride Amanda for her support, encouragement, and tolerance, and to my children Bjørn and Oline, for showing me completely different, and wonderful, sides of this life.

Contents

Part I

Overview, Background, and Discussion

Chapter 1

Introduction

> *Almost always, new software expands on previous developments; the best way to create it would seem to be by imitation, refinement and combination.*
>
> — Bertrand Meyer, 1988 [85, p. 217]

Reuse of functionality is important in virtually every software development project today, be it in the form of inclusion of open source software, calls to existing external processes in e.g. a service-oriented architecture, utilization of ready-made libraries that come with the language or platform, or reuse of internal components across various parts of the solution. In this thesis, we will focus on the kind of reuse where the reused component is directly included as an integral part of the program being developed, as opposed to the service-oriented approach.

Programming languages can be categorized in many ways, based on different properties or characteristics. One approach is to categorize them, based on the programming constructs they support, as belonging to one or more *programming paradigms*, e.g. object-oriented, functional, procedural, or logical, and typical mechanisms for reuse will differ according to these paradigms. The artefacts introduced in this thesis, as well as the discussions pertaining to them, will revolve around languages and mechanisms supporting object-oriented programming.

The basic concepts of object-orientation, including mechanisms for reuse and adaption of functionality, were initially introduced with the Simula 67 language [35] (which, in contrast to its predecessor Simula I [36], was designed as a general-purpose programming language). Simula 67 had classes and objects, (single) inheritance and virtual methods, concepts which have had a tremendous impact on the programming language community at large and have enjoyed widespread adoption.

Reuse is, arguably, tightly connected with the ability to separate different concerns, so that their implementations can be reused independently, and woven together with other components. Since the initial concepts of object-orientation were introduced, many complementary concepts have been suggested (and implemented) in order to facilitate better separation of concerns and more reuse. Examples of such concepts are virtual classes [83, 84], multiple inheritance [29, 85, 120], traits [106], mixins [26], open classes [32], aspect-oriented programming [74], subject-oriented programming [66], and generics (parametric polymorphism) [23, 91].

Along with the introduction of many programming language constructs, there has been a development of many principles, patterns and best practices, that also aim at facilitating a greater degree of reuse (as well as comprehension and ease of maintenance). Among the best-known works in this category is the book on design patterns by Gamma et al. [53]. A design pattern is a description of a component or set of components that provide a general solution to a specific, commonly recurring, problem in software development. Examples of such patterns are the Observer design pattern, in which a set of observers require notification when a set of subjects are changed in certain ways, and the Singleton design pattern, in which at most one object can be created from a given class during the life cycle of an application.

In this thesis, we will consider the reuse of (potentially hierarchical) collections of related classes that work together to solve a given problem. Such collections can be viewed as libraries or frameworks of ready-made building blocks (patterns) that can be adapted and plugged in to work with existing code.

As an example, consider a set of classes for representing a graph in the form of nodes and edges. For such graphs, there are many well-known algorithms, e.g. for searching the graph or for computing the shortest distance between two nodes. Given a general graph representation library, and another library of algorithms operating on such graphs, it would be desirable to be able to reuse both the classes representing the general graph structure and the algorithms that operate on them, and furthermore be able to apply them to related problems, such as e.g. cities that are connected by roads and railway lines, or telephone networks with centrals and lines etc. At the gist of this problem lies the issue of reusing parts that were designed and developed *independently* of each other, and being able to combine these parts in fruitful ways.

When reusing code in this manner, even for such a relatively simple and straightforward example as the one outlined above, there are several issues that are worth noting.

To begin with, it might be impossible or undesirable to change the source code of the library that is to be reused, for instance because the code is developed by an external third party. That means that even if we have access to the source code of the component, directly changing the source is problematic if the third party issues updates or bug fixes to the code. It might also be the case that the same library is reused in many places in the same application, and changing its source code to adapt to specific locations would lead to several redundant and mostly equal pieces of code. Thus, there is a need for being able to *unintrusively* and *retroactively* make the necessary adaptions to the library for the problem at hand.

In mainstream object-oriented programming languages, the approach to making such adaptions typically revolve around subclassing and overrides, or techniques such as the Adapter design pattern [53]. However, for modules/libraries containing hierarchies of classes, it might be desirable or necessary to adapt the classes at any level in this hierarchy. For instance, while behavior defined in one class often with relative ease can be overridden in a subclass in an unintrusive manner, the problem gets significantly more complex if there are several classes that refer to, or are subclasses of, each other. Making a subclass and overriding the method(s) in question, or wrapping the

class in an adapter class, will not help if other classes refer directly to the old versions. There is thus a need for *in-place*, yet unintrusive, adaption.

Reuse of independently written components often requires conformance to certain contracts or interfaces, not only structurally but also nominally (at least for statically typed languages). For instance, relating to the example above, it might be necessary for node classes to explicitly implement a certain interface in order to be used with a given graph algorithm. Meeting such demands retroactively without modifying the source code is not a trivial task in most of the mainstream object-oriented languages of today.

A final issue is that an existing library typically contains some of the functionality needed for solving a given problem, but not all of it. Thus, extension might be needed for both the set of operations and the set of classes (types) provided by the library. Retroactively creating such extensions of both sets in an unintrusive manner is what is known as the *expression problem* [134].

Package Templates (PT). The work in this thesis is based on a mechanism called Package Templates, or just PT for short. The basic ideas of the mechanism were first introduced by Krogdahl in 2001 [76] (then called Generic Packages). PT is a mechanism for reuse and adaption of class libraries in the form of templates that are instantiated before usage in a program.

The mechanism has evolved much since the initial ideas, and in Chapter 5 we will take a closer look at the PT mechanism in its current form, resulting from the work with this thesis and joint work with other researchers in the SWAT project (for more on the SWAT project, see Section 1.2 below).

1.1 Research Goals

The overall motivation behind this thesis can be summarized as follows:

> *To design, develop, and explore a mechanism for flexible code reuse and adaption of class libraries for object-oriented languages, based on the initial ideas first proposed by Krogdahl in 2001 [76].*

This goal can be broken down into two more specific sub-goals:

1. **Formulate a solid and useful design for the Package Template mechanism**. This involves carrying out research to explore the design space from the initial ideas of [76], and to experiment with and assess features in light of experience gained through programming with package templates. Furthermore, it entails to make explicit and solidify the fundamentals of the mechanism.

2. **Apply the mechanism to real-world languages, patterns and problems**. This involves coming up with implementations and specifications that relate to specific real-world languages, to formulate existing patterns and practices in terms of our mechanism, and to utilize this in order to solve real problems.

Even at this early stage, it seems obvious that these subgoals are not independent. The application of the mechanism to real languages and problems is clearly an important feedback into the research activity of formulating a design for the mechanism, while, obviously, the design is the primary input to implementations.

1.2 The SWAT Project

This thesis is part of the SWAT research project at the University of Oslo, Department of Informatics. The acronym is short for *Semantics-preserving Weaving — Advancing the Technology*. The project is funded by the Norwegian Research Council through grant number 167172/V30 as part of the STORFORSK research program. Two initial ideas were outlined in the SWAT project proposal [67]: *"Generic packages with classes that may be expanded when a package is instantiated"*, which is what has become the Package Template mechanism that is at the core of this thesis, and *"Configuration of specific models/programs based upon a model/program of a family of systems in terms of a framework class"*. The latter has been of less importance for this thesis, but more central to other students in this project (see below).

The goal of the SWAT project is to advance the state of the art for both programming and modeling languages with regard to separation of concerns and weaving/composition. The focus of this thesis is on the programming language side of the spectrum. For SWAT, an important focal point for mechanisms that weave program fragments to form new programs is that the semantics of the individual pieces should preferably be preserved to the degree possible. This entails, quoting [67], that

- *It should be possible for the programmer to easily regulate in what way an element can interrelate with other elements when it later participates in a weaving process. An element should retain its meaning during the weaving.*
- *Thus, things like name clashes, unplanned rebinding of names to new declarations, and anything that could make a set of earlier consistent elements inconsistent during a weaving should be avoided.*

Four PhD students have been involved in the SWAT project, including the author of this thesis, two of which have already graduated at the time of writing. I will briefly describe the foci of the other three students below.

Roy Grønmo — graduated 2010. The focus of Grønmo's thesis [60] is in the domain of modeling, more specifically the usage of concrete syntax (as opposed to abstract syntax) for expressing graph-based model transformations. In his thesis, Grønmo presents two main results: The first is an aspect-oriented language for UML 2 sequence diagrams that utilizes the STAIRS [68, 105] formal model for sequence diagram semantics. The second main result is an approach to define typical model transformations as graph transformations, where the developer of such transformations is utilizing the concrete syntax of the underlying modeling language.

Jon Oldevik — graduated 2010. The focus of Oldevik's thesis [98] is also in the domain of modeling. His main focus is on model composition, and mechanisms that guarantee semantic preservation in the context of such composition. The thesis presents a semantics-preserving approach to sequence aspect diagram composition, it addresses conflict and confluence in product line features, and it defines an approach for associating composition contracts for models.

Fredrik Sørensen — active. The work performed by Sørensen in the SWAT project has been focused on two main topics.

The first topic is related to the work of Grønmo on semantics-preserving weaving of UML sequence diagrams. Sørensen is a co-author of two papers that are also included in Grønmo's thesis: *A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol* [61], and, *Semantics-Based Weaving of UML Sequence Diagrams* [62].

The second topic of Sørensen's research has a great deal of thematic overlap with my own work, and is focused on the design and application of the Package Template mechanism to object-oriented programming languages. Some of the work presented in this thesis is indeed the result of the joint work of Sørensen and myself. This includes Paper I [11] and Paper V [12]; Chapter 6 gives an overview of our respective contributions to the individual papers of this thesis.

There are also a couple of papers for which Sørensen was the main author and I was a co-author that have not been included in this thesis: *Dynamic Composition with Package Templates* [111], and *Reuse and Combination with Package Templates* [112]. The reason for not including these papers in this thesis is partly that the most interesting parts have been discussed in further detail in papers that are included, and partly that some of the work contained in them is still rather unfinished.

The overall nature of our work relationship has been dialectic, and Sørensen has been an important sparring partner also for many topics for which he is not listed as a paper author.

Sørensen is also a co-author of some papers on PT where I have not been directly involved, beyond discussions on the papers' subjects in the research group. These papers are listed below:

- Sørensen and Krogdahl 2007: *Generic Packages with Expandable Classes compared with similar approaches* [123]. This paper compares an early version of the PT mechanism (at that point in time called GePEC) to similar mechanisms, with main focus on the J& language [95].

- Krogdahl, Møller-Pedersen and Sørensen 2009: *Exploring the Use of Package Templates for Flexible Re-use of Collections of Related Classes* [77]. This paper presents the main ideas from [76] in further detail, and is the first "proper" paper regarding the PT mechanism. Although it was not published until 2009, work on this paper started before I joined the SWAT project in December 2007.

1.3 Summary of Main Artefacts

This section presents the main artefacts (i.e., the main tangible outcomes) resulting from the work with this thesis, as detailed in Papers I through VII in Part II. Based on the overall goal presented in Section 1.1, we have a corresponding overall artefact:

An improved mechanism for reuse and adaption of class libraries based on the initial ideas from [76].

This artefact subsumes each of the individual artefacts below, and this work thus nicely fits with the goal of *technology research*, which Solheim and Stølen [110, p. 7] summarizes as follows: *"to make new artefacts or improve existing artefacts"* (more on that in Chapter 2); the focus of this thesis is thus on improving, extending, solidifying, and validating the initial ideas of the PT mechanism. The encompassing artefact can be divided into the following more specific artefacts:

1.3.1 PT for Statically Typed Object-Oriented Languages

This artefact revolves around the design and implementation of the PT mechanism for statically typed object-oriented languages, and is as such a direct continuation of the work initiated by Krogdahl [76]. The artefacts specific for this thesis work can be summarized by the following:

Semantics of the mechanism. This artefact consists of a set of rules for transforming code in a core subset of PT to plain Java code, thus defining the semantics of the mechanism in terms of its translation to Java.

In a wider perspective, the approach utilized for this artefact can be used as a basis for implementing PT in other statically typed mainstream OO languages as well.

This artefact is discussed in Paper VII, and builds on the foundation laid in Paper V.

A mechanism for parameterization of modules. This artefact presents a way for parameterizing modules (containing sets of classes), and thus provides parameterization at a higher level than individual classes. This idea was already introduced in [76], but the approach taken in this thesis adds significant flexibility and expressiveness.

Many of the underlying ideas and issues behind this artefact are introduced and discussed in Paper V. A further refinement of the artefact is presented in Paper VI.

Aspect-oriented programming constructs applied to PT. This artefact consists of a way to include the aspect-oriented programming (AOP) constructs of pointcut and advice in template classes, in a type safe way, and with that the ability to retroactively refine such definitions.

This artefact is introduced in Paper I, and further utilized to implement several design patterns as described in the discussion section for Paper II. AOP constructs

applied to PT is also briefly discussed as something that can be implemented by the constructs added to a version of PT for dynamic languages (see below) in Paper IV.

An implementation of PT for Java. This artefact provides a prototype implementation of PT for Java. The source code for this artefact can be downloaded from the project's software page: `http://swat.project.ifi.uio.no/software/`.

This artefact is not described in detail in any paper, but implements constructs from papers I, II, V and VI, and utilizes some of the methodologies for translation put forth in Paper VII.

1.3.2 PT for Dynamically Typed Object-Oriented Languages

PT was originally conceived as a mechanism for statically typed OO languages. However, in papers III and IV we present an approach to how PT can be applied to dynamic languages, and discuss the utility of the concepts of the PT mechanism in this context. We show how the main PT constructs for reuse, composition and adaption can be realized for a dynamic language. Specific contributions beyond the basic PT mechanism applied to dynamic languages are addressed by the sub-artefacts below.

Runtime template instantiation. This artefact presents an approach to runtime adaption and composition of entire class hierarchies, while still retaining a localized scope.

This artefact is introduced in Paper III, and further refined in Paper IV.

Meta-level manipulation of collections of classes. This artefact presents a way to utilize a meta-level protocol [75] to control the instantiation and composition of collections of classes/class hierarchies for dynamic languages. Furthermore, it presents a way for such collections to be aware of their own instantiation process, and interact with this process in meaningful ways.

This artefact is discussed in Paper IV.

An implementation of PT for Groovy. This artefact provides a prototype implementation of PT for the Groovy programming language [116], implemented in Groovy itself. It supports basic dynamic PT as well as the meta-level constructs described above. The implementation for this artefact can be downloaded from the project's software page: `http://swat.project.ifi.uio.no/software/`.

This artefact is not described in detail in any paper, but implements the constructs proposed in papers III and IV.

1.3.3 Examples and Patterns

This artefact is a library of examples that shows how the PT language can be used for developing ready-made building blocks, in the form of pluggable design patterns from [53] and an implementation of a subset of the Boost Graph Library [107]. The source

code for this artefact can be downloaded from the project's software page: `http://swat.project.ifi.uio.no/software/`.

This artefact is not described as a whole by any paper, but it is supported by the examples presented in papers I – VI, as well as the discussion sections from Chapter 6 of this thesis.

1.3.4 Requirements for the Artefacts

We will get back to requirements for the new artefacts in the form of a set of desirable properties in Chapter 4. These desiderata work together to fulfill the research goals from Section 1.1.

1.4 Thesis Structure

This thesis is divided into two main parts. Part I (the part in which the current section resides) contains an overview of the problem domain and the main results of this work. The rest of this part is structured as follows:

- **Chapter 2 – Research Method** describes the research method upon which the work presented in this thesis is based.
- **Chapter 3 – Literature Review and State of the Art** presents background material for this thesis in the form of a study of related work.
- **Chapter 4 – Problem Analysis — Basic Ideas and Desiderata** discusses the main challenges that we wish to solve in this thesis in the form of a set of desiderata for new artefacts, based on the initial ideas behind this thesis.
- **Chapter 5 – An Overview of the Package Template Mechanism** presents a high-level description of the PT mechanism as it is collectively presented in the papers in Part II of this thesis.
- **Chapter 6 – Overview of Research Papers and Discussion** presents a brief summary of each paper from Part II of this thesis. Furthermore, it contains a discussion that aims to view the paper in light of the rest of this thesis, and to address insights gained since each individual paper was written, and also in some cases include material that we could not find room for in the paper due to publication venue limitations.
- **Chapter 7 – Concluding Remarks and Future Work** concludes Part I of this thesis with a brief discussion of the main results in this thesis in light of the desiderata set forth in Chapter 4. Finally, we briefly discuss potential directions for future work.

Part II contains the full research papers. The content of the papers is identical to how they were originally published, except that the layout has been adjusted to match

the overall layout of this thesis. Some minor typographical corrections have also been made.

Table 1.1 below presents an overview of the main topics treated in each individual paper.

Paper # / Topic	I	II	III	IV	V	VI	VII
Basic PT semantics					✓		✓
Language extensions to the constructs proposed in [76, 77]	✓	✓			✓	✓	
Runtime instantiations			✓	✓			
Switchable semantic strategies				✓			
Pluggable design patterns	✓	✓	✓	✓			
Generic parameterization					✓	✓	
AOP features	✓			✓			

Table 1.1: A map of topics treated in the individual papers in Part II of this thesis.

Chapter 2

Research Method

In Chapter 1, an outline of what we have set out to research was presented. The current chapter, on the other hand, outlines how, i.e. according to which *method*, this research has been performed.

Computer science as a research discipline is relatively young, and does not have a well established set of research methods [40]. In fact, even the question of whether it indeed *is* science is debatable [37]. Interesting as that is, we shall not, however, pursue that debate any further here.

Solheim and Stølen [110] argue that much of the research in computer science departments can be classified as *technology research*. As opposed to most "classical" research, which strives to attain knowledge of the physical world, the goal of technology research is *to create artefacts which are better than those which already exist* [110]. Artefacts are objects, which may be concrete or abstract in nature, created by humans. In the field of computer science, an artefact might e.g. be a communication protocol, a sorting algorithm, a compiler, or a programming language construct.

The work behind this thesis can be described as technology research, and has focused mainly on artefacts relating to programming language constructs for object-oriented languages, including experimentation with, and evaluation of, these. The main artefacts resulting from the work with this thesis were listed in Section 1.3.

The technology researcher aims to improve existing artefacts and develop new ones by applying the scientific method through three main steps or *phases*: *problem analysis*, *innovation* and *evaluation*. It is important to note that technology research (like classical research) is an iterative process in which this method is applied repeatedly, utilizing insights gained from one iteration in the next.

Correspondingly, for this thesis it is not the case that the work started with a single analysis phase and ended with a single evaluation phase. Rather, the method has been applied many times over, not only for each paper, but typically multiple times in the research process that has led to each paper.

In the following, we will briefly outline how the steps of the scientific method have been applied to the work presented in this thesis.

2.1 Problem Analysis

The problem analysis phase of this PhD project started with an overall literature review in the field of software reuse and composition, focusing in particular on technologies for object-orient programming languages, such as e.g. various modularization constructs, aspect-oriented programming, program weaving in general, and approaches based on virtual classes, as well as a thorough investigation of the ideas in the research report on Package Templates [76] (then called "Generic Packages"). This formed the foundation for which the rest of this work is built.

For each of the papers presented in Part II, the analysis phase was typically motivated by the authors' experience with developing software in the commercial software industry, or by experience and insights gained through studying prior work and using existing research prototypes/implementations. This might then spark fresh new ideas, or bring forth an understanding of problems that were, in our opinion, not satisfactorily addressed by existing approaches.

In other words, in the terminology of [110], we identified a supposed need for new artefacts (or new knowledge, which might then be applied to produce new artefacts). For each supposed need, we initiated a focused literature review, in order to gain an understanding of the research within the problem domain. This review identified existing artefacts and approaches, as discussed in each paper's Related Work section.

Based on the analysis of existing artefacts, we had a good starting point for a problem description, and formulated an initial set of desiderata for new artefacts.

2.2 Innovation

The innovation phase aimed to realize the new ideas and address problems identified in the problem analysis phase, and resulted in new artefacts or improved (arguably) variants of existing artefacts, as described in the individual papers in Part II.

The new artefacts include new constructs in the PT mechanism, application of PT to new languages and problem domains, a better understanding and thus a better description of existing constructs, prototype implementations, and a catalog of examples.

2.3 Evaluation

The evaluation phase of the individual research papers comprises two main approaches: example studies and prototype development.

Example studies. All the papers in Section II are to a certain extent driven by the examples to which the artefacts are applied. We have described new examples, but also to a large extent reused existing examples (from the literature, and open source communities) to validate our proposals. Instances of existing examples treated in this thesis are e.g. design pattern [53, 80] implementations in papers I through IV, and the expression problem [47, 132, 134] in Paper V. A somewhat larger case study is considered in

paper VI, where a non-trivial subset of the Boost Graph Library [107] is implemented. The source code for this case study can be downloaded in its entirety by following links from the SWAT project home page: `http://swat.project.ifi.uio.no`. As a background for the discussion for Paper II in Section 6.2.2, we wrote implementations for all the design patterns from [53] in PT, and these can also be downloaded from the home page of the SWAT project.

Prototype development. The work presented in this thesis is supported by several prototypes that were created in order to illustrate, test, and validate the artefacts described in the research papers. For Paper III, we have developed a prototype implementation of PT in and for the Groovy dynamic programming language. This prototype was later expanded to support the meta-level constructs of Paper IV. A prototype implementation of PT for Java has also been developed, with the help of several master students, supporting the constructs of papers V and VI and partly utilizing the approach from Paper VII. The prototypes are available from the SWAT project home page: `http://swat.project.ifi.uio.no`

The overarching hypothesis of technology research, that we aim to test after an innovation step, is that the resulting artefacts satisfy the need(s) identified in the corresponding analysis phase [110]. Example studies and prototypes have been our main approach for checking the validity of such hypotheses, or at least for providing more support for the claim that the need(s) is (are) fulfilled. Thus, the examples and prototypes are intended to demonstrate that the artefacts developed satisfy the needs identified, and that the new or modified artefacts as such are an improvement over existing artefacts, with relation to these needs.

Chapter 3

Literature Review and State of the Art

In this chapter, we aim to present a broad background for the problem domain in which this thesis resides, in the form of a review of existing literature in the field. The selection will be based on what has been influential for our own research, and what is related to the work in this thesis. Thus, we will consider seminal works that form the pillars of our discipline, and we also aim to given an overview of the state of the art, i.e. recent works that are related to this thesis.

This chapter aims to be descriptive in nature, and will thus not focus on qualitative assessments of the respective scientific works presented below.

3.1 Separation of Concerns, Modularization and Composition.

The notion of *separation of concerns*, a phrase first coined by E.W. Dijkstra [38] (though originally used in a slightly different context), is prevalent throughout virtually all software development paradigms, methodologies, best practices and patterns. For instance, the object-oriented software development paradigm is focused on separation of concerns into classes/objects, whereas a procedural software development paradigm puts emphasis on a separation of concerns into different procedures. Software design patterns typically provide guidelines or templates for separation of concerns into different roles that for instance can be implemented as individual classes or interfaces.

Dividing a programming task into a set of independent concern implementations is typically referred to as a *decomposition* or *modularization* of the task. An inherent property of such a decomposition is obviously that the separate parts/modules need to be *composed* again for the system to be complete.

Multi-Dimensional Separation of Concerns. The kinds of decompositions and compositions applicable for any given program are highly dependent on the modularization constructs that are provided by the language/technology in which the program is implemented. In most mainstream programming languages, the programmer can choose only one such decomposition at any one time. Even though the programmer likely will strive to select the presumed best decomposition for the problem at hand,

Tarr et al. [127] point out that the fact that a single decomposition must be chosen might lead to rather arbitrary and problematic choices of decompositions of systems. Any one decomposition typically (in most mainstream programming languages) rules out any other potential decompositions of the program, and [127] calls the resulting situation *the tyranny of the dominant decomposition*, a phrase that has since been frequently cited.

Decomposing a system according to the dominant decomposition mechanism (e.g. a decomposition into a single inheritance class hierarchy as in Java [58], C# [43], etc.) arguably often leave concerns that *crosscut* the chosen decomposition *scattered* and *tangled* in the code base (since you can only have one class hierarchy in e.g. a Java or C# program). Scattering of a concern means that the implementation of a single concern is spread out through several otherwise unrelated modules/classes/functions, typically because the underlying language does not permit a single implementation of this concern given the chosen decomposition. Tangling of a concern means that the implementation of the concern is intermixed with unrelated code, leading to code that is harder to understand, maintain and refactor. Typical examples of features that often end up as *crosscutting concerns* are security, logging, transaction handling, etc.

Tarr et al. approach this problem by introducing a concept of *multi-dimensional separation of concerns* (MDSoC) through the use of so-called *hyperslices*. A hyperslice contains everything that pertains to a particular concern in the system in question. Hyperslices need not be complete according to the rules of the host formalism/programming language (and hence they cannot be semantically checked separately in context of the host language's rules), and there is thus a need for subsequent composition to form *hypermodules*. Composition is governed by composition rules (which may be written by the programmer), and these rules specify which concepts from which hyperslices should form a hypermodule.

Subject-Oriented Programming. The work on MDSoC builds, at least in part, on the research on *subject-oriented programming* (SOP) [66]. SOP is an approach that allows multiple subjects to have differing, possibly incomplete, views of objects in a system. A subject represents a view of the world, i.e. a subjective perspective, as manifested by state and available behavior. The stated primary goal of the mechanism is to *"facilitate the development and evolution of suites of cooperating applications"* [66, page 412]. Multiple subjects can be composed to allow them to interact, and to react to each other's behavior. Subject composition is governed by composition rules that may be general or specific.

In [100], a model for SOP composition rules is described, relying on *labels* that describe the declarations in a (flattened) subject. Composition clauses are then defined for such labels, which again may be used to define more high-level composition rules.

SOP and MDSoC are examples of so-called *symmetric* approaches, in the sense that the concerns of a system are all expressed using the same constructs, and composed with a common composition operator. *Asymmetric* approaches, on the other hand, typically divide a system into a base program (e.g. an object-oriented inheritance hierarchy) and a set of implementations of crosscutting concerns that are composed (or *weaved*)

with the base program according to a composition specification.

Aspect-Oriented Programming. Specifically designed for dealing with crosscutting concerns, *aspect-oriented programming* (AOP) [74] is an example of an asymmetric approach. With AOP, a program in e.g. plain Java represents the (hopefully well-structured) base program, and *aspects* represent the crosscutting concerns that would otherwise be scattered and tangled throughout the modules of the base program. For Java, AspectJ [5, 33] is a popular tool for performing such compositions of base and aspect code.

Aspects are typically represented by a syntactic construct that includes *pointcut*s and *advice*s. A pointcut is a specification of locations in the base program at which the aspect should be applied. A pointcut thus represents a set of *join points*. A join point is a location in the base program where advice can be applied. The set of available join points is based on the underlying semantic model of the target language, and can for instance include method calls, field writes, runtime exceptions, etc. This set of join points is often referred to as the *join point model* of the AOP mechanism.

According to Filman and Friedman [51], the essence of aspect-oriented programming is *quantification* and *obliviousness*. Quantification is what is achieved by using pointcuts – a single aspect can refer to (and affect) many distinct locations in a given base program by specifying a query or predicate over its semantic elements. Obliviousness entails that the base program is unaware (oblivious) of the fact that aspects might modify it to interfere with its execution.[1]

A criticism of many aspect-oriented programming constructs is the fragility of the pointcut definitions [119], in the sense that seemingly innocuous refactorings of a program, such as e.g. method renames, can render aspects inapplicable and, as a consequence, entire programs non-functional. Coupled with the obliviousness of the base program with regard to potential aspects, and AOP's disregard for standard OO encapsulation principles, this presents a challenge for developers utilizing AOP mechanisms. Certain approaches, such as e.g. PostSharp [117] circumvent such issues by instead relying on explicit join point annotations in the base program. However, the base program is then no longer oblivious of the fact that aspects might, and most likely will, interfere with its execution at specific locations in the code. Whether this lack of obliviousness is a drawback or an advantage is still largely a matter of debate in the AOP community.

The join points of a language can be seen as implicit events from an event-driven programming perspective. This is utilized in e.g. the EScala language [55], which extends the Scala language [96] with imperative C# -style events and declarative implicit events. EScala also supports event composition.

[1] Saying that a program is *aware* or *unaware/oblivious* of anything is obviously just a figure of speech, as the program itself is not in any way sentient. The underlying meaning is that the program does not contain any references to the aspects, and one cannot deduce from looking at the base program alone whether it will be advised by aspects or not. However, note that while the base program is oblivious of the aspects, the programmer writing the base program may obviously be aware, and might even be designing the program knowing, and planning for, that aspects will be applied to it.

Feature-Oriented Programming. With *feature-oriented programming* (FOP) [102], the programmer can develop a set of independent *features*, which can subsequently be composed to form new objects. Thus, features, and not classes, form the "blueprint" for creating objects in FOP.

The *AHEAD* mechanism and its accompanying *AHEAD Tool Suite* (ATS) [15, 16] is a general mechanism for synthesizing programs by composing incremental software features. ATS allows features to be successively refined in separate files, and provides a strategy for recursively composing features from such files at the source code level. The composed code may then be processed by e.g. an ordinary Java compiler. ATS does not provide any guarantees with respect to preserving original bindings in the composed result.

While the focus in this thesis is on programs and programming language constructs, many FOP mechanisms allow features to contain other types of information as well, such as for instance documentation, HTML files, etc., which can be composed using the same constructs as for source code features.

Traits and Mixins. The *Traits* mechanism [106] attempts to better facilitate reuse by breaking the source code into units smaller than individual classes. The main idea is that the basic unit of composition is a stateless collection of methods called a *trait*, representing a reusable implementation of a concern. Traits may depend on functionality offered by other traits, and the mechanism has constructs for expressing both provided (implemented) and required methods. Both new traits and classes can be composed from other traits, and when a class is composed all the required methods must be provided either directly by the class itself or by traits used to define the class.

Trait composition is orthogonal to regular single inheritance. However, there is no inheritance between traits, only between classes. The traits involved in a composition forming a class or another trait are said to be *flattened*; this means that (1) the trait ordering in each composition is irrelevant, and (2) that a class composed from traits is semantically equal to a class in which all the methods are defined directly in the class. Traits were originally developed for the Smalltalk dialect Squeak [30], and supports method name aliasing and method exclusion upon composition. A statically typed version also exists [92].

Although traits were originally defined as stateless method collections, more recent work [19] has shown how a stateful variant may be designed and formalized.

Mixins [26] are similar in scope to traits in that they target the reuse of smaller units that are composed into a class. Mixins also define provided and required functionality, and the main difference between them and traits is the method of composition. Mixins traditionally rely on (single) inheritance, by defining a mixin as a subclass with an as-of-yet undefined ("virtual") superclass; mixins are linearly composed along this inheritance hierarchy. Calls to methods in the superclass are allowed through `super` references. The actual superclass is specified (by the programmer) when the mixin class is used in a program.

Variants of mixins and traits, with varying syntax and semantics, are implemented in several widely used languages and mechanisms, such as e.g. Scala [96], JavaFX [42,

50], Ruby [128] and PHP [1].

A recent addition to the family of mixin-based languages is Magda [22]. The following features differentiate Magda from otherwise similar languages: Every object is constructed from one or more mixins (i.e. there is no separate notion of a class); every field or method reference is qualified with the name of the mixin in which it was originally declared, thus enforcing *hygiene*; the language employs a novel constructor/initialization scheme that alleviates the need for an exponential number of constructors when combining mixins.

Virtual Classes. The idea that not only methods, but also classes, can be virtual was pioneered by the BETA language [83, 84], and has since been the foundation for and focus of a great deal of research. In a language supporting virtual classes, inner (nested) classes of an outer class may be defined as virtual, meaning that they can be overridden (extended) in subclasses of the outer class, thus constituting significant variability points.

Family polymorphism [46] is a programming language feature based on virtual classes that allows relations (in the form of type hierarchies) between *families* of classes to be expressed and statically type checked, so that objects from different families are never mixed. Families are expressed through the use of nested classes, where the outer class, or an object of the outer class (depending on the mechanism in question), represents the family itself. The inner (usually virtual) classes, or objects of these, represent the members of the family. The outer class can typically be subclassed, and the inner classes can then also subsequently be refined through subclassing. Family polymorphism allows the families to be treated polymorphically (i.e. both the superclass family and the subclass family can e.g. be handled by the same method, where the exact runtime type of the family is not known statically), and dependent types (types that depend on a value) are used to ensure static safety.

The gbeta language [45], which builds on the BETA language, implements type safe family polymorphism as one of its core contributions, by introducing some restrictions with regard to final binding.

J& [95] supports family polymorphism in a Java-like language, and also supports a limited form of multiple inheritance, making it well-suited to solve problems that relies on combining independent extensions to a base library, like e.g. the expression problem [134]. However, this approach also entails that such combinations must be planned in advance for the composition to succeed.

NewSpeak [28] is a quite recent language based on virtual classes, where *every* class reference (including superclass references) is virtual. This entails that every class can function as a mixin. Furthermore, the *only* modularity construct (besides methods) is the class, and class nesting is applied to provide a module hierarchy. A module can be adapted and extended by creating subclasses of outer and inner classes, and by overriding virtual definitions. The fact that NewSpeak is dynamically typed is of course a large part of what allows its wide-ranging flexibility. (However, a type system based on *pluggable types* [24] is planned for a future version of the language [25].)

The Caesar language [87, 88] has some novel characteristics in the form of a mix of

virtual class-based and aspect-oriented programming-based features, while also supporting mixin composition and merging of virtual class hierarchies through linearization. Central to the Caesar approach are *collaboration interfaces*, that support mutually recursive nested virtual types and lets the developer specify provided and required operations. The main implementation of Caesar is CaesarJ for Java [4].

Collaboration interfaces were introduced in [86]. They allow for separate classes to *implement* the interface (i.e. implementing the provided methods of the interface) and *bind* the interface (i.e. implementing the required methods of the interface). By using constructor arguments to wrap classes, one can adapt existing class hierarchies to new interfaces. The paper also describes wrapper recycling, in order to prevent a multitude of wrappers for the same wrapped objects.

ECeasarJ [41, 93] is a more recent extension to CaesarJ that includes event handling, both for explicit imperative event signaling and for declaratively defined events. Events can be composed using mixin composition, relying on a linearization order to resolve conflicts and to determine the order of `super` events. State machines can be included directly in classes, and utilized to react to events.

3.2 Retroactive Adaption, Composition and Implementation

The use of readymade components is prevalent throughout most of the field of software development, as such components present a large potential for productivity boosts. However, one issue with using such components is that they do not always perfectly match the problem at hand. Furthermore, even if they from a functional and structural standpoint *do* match perfectly, there might be explicit formal demands in existing code (such as e.g. requirements for explicit nominal interface implementations) that prevent reuse.

At the gist of this issue is the problem of *retroactive adaption*. That is, how can we *unintrusively* adapt an existing component (class, class hierarchy, etc) to a specific use case (without changing its original definition, which typically exists in source code form). A related issue is how to perform such adaptions without affecting other clients of the code that are oblivious to the change.

A common, though perhaps somewhat cumbersome, approach is to apply the Adapter design pattern [53] to enclose the original component in a wrapper that adapts it to the new requirements, typically by delegating operations through to the original component. However, this can lead to object identity issues (often colloquially referred to as *object identity hell* [69, 86]), problems of code duplication, reduced type safety necessitating runtime type casts, reduced execution speed, etc. Especially cumbersome and inconvenient is the Adapter pattern and related approaches when dealing with adaption of a set of cooperating classes or class hierarchies.

Recent versions of the C# language include a construct called *extension methods* [89, sec. 10.6.9]. This allows classes and interfaces to appear as having been extended with new methods without changing their original source code (or even recompiling

them). However, extension methods are really just syntactic sugar for static methods, effectively precluding e.g. method overrides and dynamic dispatch.

There is also a proposal for including a form of "extension methods" in Java [57] (there called *default* or *defender* methods), however with a slightly different aim: to allow the modification of interfaces after their initial publication (for instance in the Java class library), without breaking existing clients. Extension methods allow interfaces to be retroactively extended without requiring modification to existing implementors by providing default method implementations. In contrast to the C# variant, the Java extension method proposal allows for method overriding.

Classboxes [20] allows methods to be retroactively added to existing classes, and replacement of existing methods in existing classes, within the scope of a *classbox*. A classbox must be explicitly imported by client code for its modifications to take effect. The original paper and implementation presented the mechanism as an extension to the (dynamic) Squeak programming language [30]. A later paper [18] shows how classboxes can be applied to the (statically typed) Java language.

Expanders [135] is a language construct for object-oriented languages that allows static addition of new methods and state to existing classes in a retroactive manner. The expansions are utilized explicitly by client code, allowing unintrusive expansion as well as differing views of classes between clients. Expanders for Java are realized by the eJava language, and implemented through a translation to pure Java code utilizing the Polyglot extensible compiler [94]. The expansions are implemented as wrappers around objects of the expanded class. An expander may override the methods of another expander in the same family, but cannot override ordinary methods in the expanded class.

JavaGI [136] is a system inspired by the type classes [64] of Haskell, aimed at achieving a similar flexibility in Java. Like with type classes, JavaGI allows for retroactive interface implementations to be added to existing classes in a non-intrusive manner. Utilizing JavaGI's *multi-headed interfaces* makes it possible for several classes to jointly implement a concept, and additionally provides support for multiple dispatch and family polymorphism.

The Scala language [96] "*fuses object-oriented and functional programming*" [97] in a statically typed context. This allows it to support a concept called *implicits*, which can be used to emulate type classes and thus achieve functionality that is similar to extending existing classes and objects with new functionality in a retroactive fashion [99]. However, since implicits are not true extensions of classes, they do not support dynamic dispatch, and might thus be a somewhat awkward fit for object-oriented programs. Furthermore, due to implicit conversions from one type to another, problems with object identity might easily occur.

Some approaches to aspect-oriented programming support retroactive adaption of classes and interfaces. An example of this are the *inter-type declarations* of AspectJ [5]. Using inter-type declarations, it is possible to add methods and fields to existing classes or interfaces, and to add new interface implementation declarations or superclass declarations.

MultiJava [32] adds *open classes* and *multiple dispatch* [90] to the Java programming

language. Open classes makes it possible to add methods and fields to any class retroactively, without changing the original declaration. Clients of open classes explicitly import packages containing added methods and fields.

Arguably, multiple dispatch is a natural companion to open classes, since it allows methods to perform dynamic (runtime) dispatch on all method parameters, thereby allowing new subclasses to influence the dispatch of calls to existing methods with new overrides.

3.3 Module Adaption and Composition in Dynamic Languages

Dynamic programming languages have garnered increasing popularity in the last few years [133], and proponents claim that the expressivity they typically provide can more than make up for their relative lack of compile-time safety. With respect to composition and adaption, dynamic languages typically offer constructs that exceed those of their statically type checked counterparts, supporting features such as open classes or "monkey patching" (runtime modification of classes and objects), meta-object protocols, runtime evaluation of code in the form of strings, etc.

Runtime class modification. Changes to classes at runtime are utilized to a large extent in many popular mainstream libraries for dynamic languages, such as Rails [104], Grails [115] and Django [52], in order to add framework-specific functionality to shared classes. An example of that is how the Rails framework adds methods to the `Integer` and `String` classes of Ruby in order to support statements such as e.g. `40.years.ago`[2] or `"Hello World!".as_json`[3]. A problem with this approach is that such changes are *global* in scope, meaning that the frameworks perform runtime modification of public state, which since long has been recognized as a generally bad idea [138].

Classboxes, as mentioned above, are one approach to handling this issue in a more controlled manner.

Meta-object protocols. In their seminal 1991 book [75], Kiczales and Rivieres present a convincing case for incorporating meta-object protocols (MOPs) in programming languages. With a MOP, the semantics of object-oriented programming language constructs are made available to the programmer for manipulation, so that it is possible to redefine, typically at runtime, what a base program actually means. Such concepts have found their way into many programming languages, and it is especially common for dynamic languages to possess such capabilities.

[2]See `https://github.com/rails/rails/blob/master/activesupport/lib/active_support/core_ext/integer/time.rb` for implementation.

[3]See `https://github.com/rails/rails/blob/master/activesupport/lib/active_support/json/encoding.rb` for implementation.

Also for aspect-oriented programming (AOP), ways to control the semantics of AOP constructs at runtime have been addressed for dynamic languages, e.g. by Tanter (with special focus on scoping semantics) [124, 125] and Dinkelaker et al. (meta-*aspect* protocols) [39].

3.4 Generic Programming and Adaption

Generic programming, also often referred to as parametric polymorphism, is an approach to programming where algorithms, classes or other units of modularization are written in such a way that they depend on a selection of abstract qualities from (some of) the types upon which they depend, instead of being dependent only on concrete types. This can provide flexibility and a loose coupling between different parts of a program, while still retaining type safety, and can thus make components more easily reusable.

Generic templates. A template in the context of programming languages is a blueprint for creating units of code in the language. Such blueprints can be *instantiated*, thus forming actual units of code. The templates can be syntactically and semantically checked entities in their own right, with well-defined semantics, or they can be macro-like constructs intended for plain preprocessing of text, depending on the underlying language/mechanism.

A generic template is a template with parameters that can be supplied (either explicitly or implicitly) when instantiating it, thus creating specific variations from the blueprint.

The C++ *language* has long supported generic programming through its templating mechanism. C++ templates can be used to create generic function or class definitions in a flexible manner, and such functions and classes are reified as concrete functions or classes at compile-time based on their parameters. I.e., the generic mechanism of C++ produces separate (heterogenous) code for each distinct generic instantiation, and templates are not compiled before they are instantiated in a program, as opposed to generic classes in e.g. C# [43]. Furthermore, there are currently no good ways (beyond comments in the source code) to specify requirements or bounds for generic parameters in C++, thus making separate type checking of templates impossible. This has the advantage that it provides maximum flexibility with regard to different (and potentially unanticipated) parameterizations, but at the same time it presents a problem in terms of robustness and blame assignment when generic code fails to compile.

While not allowing explicit bounds for template parameters, C++ does allow templates to be specialized with respect to their parameter(s). This entails that it is possible for a programmer to explicitly write different versions of a template for different parameterizations. For instance, one may write a specialization for the case where a parameter is the type `char`, or one may e.g. write a *partial* specialization (that still allows further parameterization) where the parameter is any pointer type. The main reason for writing such specializations in C++ is typically to achieve better performance.

The *Ada programming language* supported, from early on [81], generic procedures and packages, collectively referred to as *units*. Generic units in Ada must be instantiated before they can be used, and the formal generic parameters can be both types and values. In contrast to C++, generic units in Ada can be fully typed checked before instantiation.

More recent versions of Ada [14, 48] have added support for object-oriented programming.

Concepts. A *concept* is, in the context of generic programming, a set of requirements consisting of required operations (methods) and data type constraints. A type (or a set of types) is said to *model* a concept if it fulfills these requirements. A *concept map* is then a specification that describes *how* a type (or a set of types) models a concept. Concept maps can be explicit or implicit/automatic. Since there is no explicit language support for expressing generic constraints in C++ templates, quite a lot of research has been focussing on how to incorporate a notion of concepts that can be checked by the compiler in the language.

The $\mathcal{G}$ language [108] compiles to C++ and contains explicit support for constraining generic code in the form of concepts and models. Based on this, $\mathcal{G}$ can support modular type checking of generic code. $\mathcal{G}$ also performs separate compilation of templates, distinguishing it from the inclusion-based compilation model of C++.

ConceptC++ [59] partly builds on the work on $\mathcal{G}$. The language supports explicit generic constraints through language constructs for concept definitions and constrained function and class templates, but utilizes the inclusion-based compilation model of standard C++. A stated goal for ConceptC++ was to form the basis for the inclusion of concepts in the (then) new C++0x standard. However, support for generic constraints through explicit concepts was dropped from the latest (as of August 2012) version of C++ [121]. An interesting discussion on the underlying issues that led to this decision can be found in the note *Simplifying the use of concepts* by Stroustrup [122]. The main tension seems to be between structural versus nominal typing, or implicit versus explicit concept maps.

As mentioned above, the Scala language supports *implicit* declarations [99]. By using implicits one can implement a form of concept maps by specifying implicit conversions between types. However, problems with object identity might limit the usefulness of implicits in this context.

JavaGI [136] was discussed briefly above, in the context of retroactive adaption. Generic programming is obviously suited for creating variation points that can be exploited in a retroactive fashion by new clients. JavaGI vastly expands the possibilities for generic programming and expression of generic concepts in Java, adding support for constrained interface implementations (e.g. implementing a `compare` operation for a list class where the element type of the list is constrained to implement the `java.lang.Comparable` interface), self types (binary methods), and bounded existential types (allowing expression of types that are part of multi-type concepts).

cJ [72] allows different *parts* of a class to be conditionally implemented based on whether a particular type parameter fulfills certain constraints or not. The authors

demonstrate the utility of this approach by a refactoring of the Java collection classes, and explain how a type safe implementation that would otherwise result in a combinatorial explosion of classes can be created in a backwards compatible manner through erasure [27].

Virtual types. An alternative to traditional type parameters as found e.g. in Java and C# is to instead use *virtual types* to represent this kind of variability. A virtual type is an abstract type specification that can be bound to a concrete type or further refined in a subclass of the defining class. Virtual type definitions thus *propagate* from a defining class to its subclasses, in the same way as fields and methods do, and as opposed to traditional type parameters.

Thorup argued [129] that the inclusion of virtual types in Java would be a better match for the language than the type parameters that ultimately found their way into the language standard [58]. In Thorup's proposal, Java classes can contain bounded type definitions (`typedef`s), syntactically akin to those of C++. These definitions can be used to define generic classes that abstract over an open set of types.

A problem with Thorup's approach is that static type safety is reduced (more runtime tests are needed), due to allowing covariant overrides of method signatures (which happens automatically when the programmer binds a virtual type to a subtype of its bound). Thus every class could now potentially be subject to runtime exceptions due to covariant subtyping, in the same way the arrays in Java already are.

In [131], Torgersen shows how an approach to genericity based on virtual types can be made statically type safe. The essence of this approach is *final binding*, i.e. that a virtual type can be bound once and for all in a class, so that it cannot be further bound in its subclasses. Furthermore, method parameters (and the `this`/`self` reference to the current object instance) must be constant/final.

A subsequent paper by Thorup and Torgersen [130] presents an approach that combines virtual types, in a type safe variant, with structural subtyping for a more flexible solution. This gives three "dimensions" of subtyping: the ordinary subclass variant, covariance of generic parameters (i.e., `Set[E <: Car]`, where `<:` is the subtyping relation, is a subtype of `Set[E <: Vehicle]`, however, `Set[E = Car]`, where `=` binds a generic parameter to a concrete type, is **not** a subtype of `Set[E = Vehicle]`), and binding of generic parameters to their bound. This allows virtual types to be used in many situations where parameterized types traditionally have been considered a better option.

Chapter 4

Problem Analysis – Basic Ideas and Desiderata

The main motivation behind the Package Templates (PT) mechanism, and thus the starting point for this thesis, is to better facilitate reuse and adaption of modules of code for object-oriented programming languages. In the rest of this chapter, we discuss in further detail what this entails, and present a list of desiderata that the work in this thesis should aim to fulfill. The basic ideas behind the PT mechanism, which can be seen as a frame of reference for our desiderata, are explained briefly in Section 4.1 below. Where relevant, we relate the desiderata to existing mechanisms as presented in the previous chapter.

Note that existing mechanisms and related work may well fulfill some of the desiderata listed below, but not, as far as we know, in such a way that the totality of our desiderata is met by a single, unified, approach. The latter is what we hope to at least come closer to with this work.

The list of desirable properties is based on an understanding of the problem domain that we have reached through studying related work as presented in Section 3, through programming and analysis of examples and case studies, through the author's practical experience with developing software in an industrial context, and through our work with the papers included in this thesis. As such the list of desiderata is not independent from the work presented in the papers; to the contrary, the desiderata have changed with our experience and our work with the thesis, as our understanding of the problem domain has grown. That is not to say that there is a direct correspondence between the results attained in each paper and the desiderata presented below; some of the desiderata are indeed not completely fulfilled by this thesis, and may thus be addressed further in future work.

4.1 Basic Ideas Behind the Package Template Mechanism

The work in this thesis is, as mentioned in the introduction, based on the initial ideas for the Package Template (PT) mechanism by Krogdahl et al. [76] (at that point in time called GePEC). Here, we will briefly summarize the main points of this initial descrip-

tion of PT, as a background for further discussion, and as a frame of reference for the desiderata we present below. A more complete overview of PT, with both the initial ideas and new developments stemming from the work with this thesis, can be found in Chapter 5.

An important inspiration for the PT mechanism was *virtual classes* [83, 84]. Languages that support virtual classes allow the programmer to specify an outer class, and several inner (usually virtual) classes. The virtual inner classes can subsequently be collectively refined (overridden) in subclasses of the outer class, creating quite a flexible system. However, this flexibility comes at a cost of a somewhat complex type system (if static type safety is required/desired when using refined virtual classes — se e.g. [131]). Thus, an initial goal for PT was to create a system that can achieve a lot of what can be achieved with virtual classes, but with a simpler type system.

One approach towards attaining this simplicity is to remove one level of dynamic (runtime) nesting, by removing the outer class. In PT, instead of using an enclosing outer class, modules are defined in the form of *templates*, which can contain collections of classes and interfaces.

Before the classes in a template can be used in a program, the template must be *instantiated*. The result of such an instantiation is that a new copy of the template's contents is created, and inserted into the program. Using the terminology of macros, one might say that the template is *expanded*, and the PT approach is indeed somewhat similar to that of macros. However, it is important to note that, as opposed to typical textual macros, templates in PT can be syntactically and semantically checked as separate entities.

Each instantiation of a template creates a new copy of the template's contents, and multiple instantiations of a given template will thus result in multiple distinct sets of classes.

A central feature is that the template's contents may, for each instantiation, be collectively adapted to the problem at hand. Such adaption is supported by PT through renaming of classes, methods, and fields, refinement in the form of overrides and addition of new methods and fields, and class merging.

The list of desiderata presented in the following sections embraces and extends the original ideas from [76]. That is, it includes both properties that were not targeted by the approach from [76], as well as properties that, at least to a certain extent, were already fulfilled by the initial description of the basic PT ideas. However, these latter desiderata are still relevant for the work in this thesis as well, because they should be fulfilled (or upheld) also for any new semantics, approaches, or constructs introduced in the papers in Part II of this thesis.

4.2 General Desiderata

In this section we discuss desiderata that are general in the sense that they can apply to many different mechanisms, and also in the sense that they are not explicitly tied to the topics discussed for related work in the previous chapter.

Desideratum 1: ***Applicability to more than one language.*** The core constructs of the mechanism should be designed so that they can be added to a variety of object-oriented programming languages.

Discussion. Being applicable to more than one language entails that the mechanism cannot be strongly bound to the concepts of any given language, beyond what can be considered as "standard" for OO languages, such as e.g. classes and objects, inheritance, polymorphism, and encapsulation. Furthermore, it also entails that the mechanism cannot be something that has utility just because of idiosyncrasies in a specific language.

When adding the mechanism to a new target language, one will have to be prepared to adapt the syntax and semantics of the mechanism, in order to provide a "good fit" for the target language. It is desirable that the mechanism appears as "native" to the target language as possible.

Desideratum 2: ***Low performance overhead.*** The runtime performance overhead of applying the mechanism should be acceptable for use in real-world scenarios.

Discussion. This desideratum has two distinct implications: 1) With regard to the mechanism itself, it should be designed in such a way that it is possible (and preferably relatively straight-forward) to create an implementation that produces efficient code in the target language. 2) An actual implementation should produce programs with acceptable performance.

Obviously, what is deemed an acceptable performance overhead will vary with different scenarios, problems and languages, spanning from zero runtime overhead to a noticeable runtime lag. However, a natural target for such a desideratum would be to try approach the runtime performance of the underlying host language, i.e. to aim for *no* runtime overhead compared to code written directly inline, using only the native constructs of the target language, for the same purpose. It seems clear, however, that there are certain scenarios in which this will not be possible, or at least very hard, to attain with an implementation of the PT mechanism.

Desideratum 3: ***Applicability to real-world problems.*** There should be a battery of examples that show how the mechanism can be applied to, and be useful for, real-world problems and programs.

Discussion. Applicability to real-world problems paired with the fact that the mechanism is intended to be incorporated into *existing* languages (as opposed to defining a new language in and of itself), necessitates that the mechanism is able to make use of, and extend, existing libraries and frameworks defined in the target (host) language. This entails that there should be a correspondence between the constructs of the mechanism and constructs in the target language, or, alternatively, that the constructs can be implemented directly within the target language.

On a more mundane level, this desideratum also necessitates that at least one implementation of the mechanism should exist.

Desideratum 4: ***Specifications.*** There should be specifications that describe the syntax and semantics of the mechanism, in the context of specific, real-world, target languages.

Discussion. As stated in Desideratum 1, we intend for our mechanism to be applicable to a variety of object-oriented programming languages. This entails that the overall mechanism in itself is more of a set of ideas than something that can be described by a formal specification. Rather, each particular *target language implementation* of the mechanism will require its *own* specification (possibly stemming from a common base specification). This is both because of the interaction between the semantics of the mechanism and the semantics of the target language, and also because each distinct target language will require distinct approaches to make the mechanism best suitable for that language.

4.3 Specific Desiderata

In this section, we discuss desiderata for the PT mechanism in relation to the topics discussed for related work in the previous chapter. For each section in Chapter 3, we have a corresponding section below (with the same title), in which we discuss specific desiderata for our mechanism pertaining to that topic.

4.3.1 Separation of Concerns, Modularization and Composition

Desideratum 5: ***Semantics preservation with respect to composition.*** The composition operations should preserve the semantics of the composed elements.

Discussion. A system can be said to be semantics preserving with respect to composition if it preserves selected semantic properties of the parts that are to be composed in the composed result. In our context, this specifically means that the static semantics, i.e. the *bindings* of references to declarations, should be preserved when (previously unrelated) elements, in the form of templates and template classes and interfaces, are composed.

At the outset, supporting semantics preservation obviously entails that it must be possible to define the semantics of individual components that are applicable for composition without taking any actual composition into account. In other words, it should be possible to establish both the static correctness in terms of types and bindings, and, subsequently, the meaning of a component in isolation, without considering potential use cases. However, requiring that a component in itself is declarationally complete puts some limitations on the kinds of components that can be expressed, and the flexibility with which such components can be reused.

The templating mechanism of C++ is one example of a widely used mechanism that favors expressivity and flexibility over semantic checkability. The lack of semantic checks is mainly due to the fact that the parameters to the templates do not have any explicit constraints, and so the types (and thus consequently also the bindings) within a C++ template cannot be established independently of the template's usage, as mentioned in Chapter 3. Several works have sought to address this by adding constraints to the template mechanism, e.g. ConceptC++ [59] and the $\mathcal{G}$ language [108]. Like these approaches, we want to provide separate semantic checkability of independent modules in our mechanism, while at the same time giving up as little flexibility as possible.

However, even if the bindings of a component in source code form can be established independently of its usage, this is clearly not a guarantee for the preservation of these bindings in the composed result. An instance of this can be seen in implementations of feature-oriented programming (FOP) [3, 102] as applied to the Java language. There, the composed semantics may depend on the (potentially unplanned) interaction of elements in the composed parts.

Correspondingly for Traits [19, 92, 106], the semantics of a class composed by individual traits is defined to be the same as if all the pieces of code in the individual traits were written directly within the composed class (this is known as the *flattening property* of traits). It is perhaps not obvious that this, in general, does not preserve the semantics of the elements in a composition. In fact, a language that allows method overloads based on formal parameter types (in the same way as e.g. Java does) cannot support both semantics preservation of trait code and the flattening property at the same time. Consider the simple example below, where the syntax is intended to represent a language resembling Java, only with traits:

```
trait T1 {
   void m(Object o) { System.out.println(o); }
   void f() { m("Hello world!"); }
}

trait T2 {
   void m(String s) { throw new Exception(); }
}

// C is defined by the composition of T1 and T2
class C with-traits T1, T2 { }
```

In the example above, for the flattening property to be upheld, the call to `m` in `f` needs to resolve to the signature that has a `String` as its formal argument type, while with semantics preservation, we actually want the opposite result, i.e. that the call to `m` in `f` should still resolve to the overload that takes an `Object` as its formal argument type.

So, to summarize: a goal for this desideratum is to preserve much of the compositional flexibility from mechanisms like e.g. C++ templates, Traits and FOP, while retaining the semantics of the individual components in the composition.

Desideratum 6: ***Composition of previously unrelated components.*** The mechanism should support composition of elements that were not explicitly developed with their mutual composition in mind.

Discussion. The ability to compose previously unrelated program components is important in order to maximize the potential for reuse of components from different sources (otherwise, only components with an explicit, priorly established, relationship would be composable, effectively limiting composition to components stemming from the same source). This means that it should not be necessary for two components to explicitly state a mutual relationship for them to be composable, neither should it be required that they stem from a common source.

Furthermore, composition of unrelated components entails that the mechanism should have constructs for dealing with conflicts that might occur when composing such elements, for instance in terms of precedence rules or explicit conflict resolution constructs available to the programmer, consequently allowing the developer to adapt the conflicting elements so that the composition can succeed.

When supporting such compositions of unrelated elements in an object-oriented programming language, there are several options for the granularity at which elements can be composed.

With Traits, Mixins, and similar approaches, classes can be composed from (potentially incomplete) class fragments. Mixin implementations typically rely on a linearized inheritance chain to *mix in* the different class fragments, while traits, as mentioned, utilize a flattening composition. Traits support exclusion and name aliasing to deal with conflicts.

Moving up one level of granularity, new classes can be composed by utilizing existing (complete) classes. Ordinary inheritance is probably the most wide-spread approach to class composition; with single inheritance the code of the parent class is "composed" with that of the subclass.

With multiple inheritance the expressiveness of this mechanism is increased; any class can state that it is a subclass of a set of other classes in languages that support it (given that certain language-specific rules are adhered to). Using multiple inheritance, the composition of class fragments can to some degree be emulated (e.g. an abstract class with a collection of methods can be considered a mixin, depending on the inheritance semantics of the language). Some languages, like e.g. Eiffel [44], even provide means for conflict resolution for instance in the form of renaming or precedence rules. Still, multiple inheritance comes with a set of well-known problems of both technical and conceptual nature, the most significant of which is probably the *diamond problem* [109]. This problem is part of the reason why traits traditionally only support *stateless* code. When multiple inheritance is used with a primary purpose of code composition for reuse (as opposed to for defining type hierarchies), we believe that a more static approach to code reuse can be taken, so that the need for multiple inheritance reified as a runtime type hierarchy is greatly reduced.

In this thesis, collections of classes are our main focus. When collections of classes are composed, it should be possible to compose two or more classes *in a single con-*

ceptual operation, necessitating that relations between multiple classes are taken into account in the composition. Such relations include inheritance and interface implementations, as well as signatures typed with classes that take part in the composition. In such compositions, we want to support stateful (as well as stateless) classes.

In the J& language [95], the programmer can utilize family polymorphism combined with multiple inheritance of the outer enclosing class to create and compose different extensions to a common base library. However, this implies that the classes we want to compose must already be related through their parent class. We want similar capabilities, but, as indicated above, without this restriction.

A final step up the granularity level brings us to the global scope. Composing elements from a global point of view entails that every class in an entire program is a potential target for composition. This is the approach typically taken by aspect-oriented programming [74], where pointcuts range over classes with which code in the form of advice is composed. However, several works (e.g. [87, 119, 124]) have identified the need for a more controlled approach, and we agree with this. Thus, we will not set out to support composition e.g. through potentially wide-ranging pointcuts with wild-cards.

Desideratum 7: ***Isolated adaption and composition.*** Changes to, and compositions involving, an element should be isolated from the rest of the program.

Discussion. Isolation in this context means that if an element is adapted, refined, or composed with another (cf. Desideratum 6, and the forthcoming desiderata 8 and 9), the scope of these changes should be explicit (and not global), so that unrelated parts of the program are not unintentionally affected. In other words, the static semantics of unrelated pieces of code should remain unchanged.

The requirement of isolated adaption and composition is thus strongly related to the above requirement concerning semantics preservation; if one does not have such isolation, the semantics of the program as a whole can be compromised. Furthermore, isolation in this context also entails that several in themselves conflicting adaptions and/or compositions involving a given element should be allowed as long as they are done in separate scopes. This is something that is not easily achieved with global approaches, such as e.g. AspectJ and similar technologies.

4.3.2 Retroactive Adaption, Composition and Implementation

When the term *retroactive* is used as a qualifier for an action (e.g. adaption, composition, or implementation) in our context, it means that this action takes place after the point in time when a component was originally written. Thus, retroactive adaption, composition and (interface) implementation implies that such operations should be supported for *existing* components, without changing their source code (i.e., the action should be *unintrusive*).

Desideratum 8: ***Name changes of program elements.*** It should be possible to modify the names of the program elements in a composition so that new names replace all occurrences of the old names (based on established bindings, cf. Desideratum 5) in the composed result.

Discussion. As mentioned for Desideratum 6, supporting composition of program elements requires a mechanism for conflict resolution. One way to support this is through explicit renaming of conflicting elements. This would require support for renaming at least fields, methods and types (classes, interfaces).

Conflicts aside, renaming of elements can be vital also to the overall comprehensibility of a program [49, 70]. Providing names that are meaningful in the context of the application being developed, and not just in the isolated context of the library being *(re)used* in the application, is something that is often approached through e.g. adaptors and delegating objects, but which can be approached more directly through renaming.

Furthermore, renaming can make the co-utilization of originally incompatible prewritten libraries possible, by adapting one library to the requirements of the other, or vice versa (i.e., adapting the *requirements* to the signatures of an existing library).

Desideratum 9: ***Refinement at any level.*** It should be possible to retroactively refine existing definitions, in-place, at any level in an inheritance hierarchy, i.e., not just at the leaf nodes.

Discussion. In mainstream statically typed object-oriented languages, refinement of classes is typically possible only by adding new leaf nodes to the inheritance tree (or directed acyclic graph, in the case of multiple inheritance). As an example, consider the following small class hierarchy of classes for defining graphs.

```
class GraphComponent { }
class Node extends GraphComponent { }
class Edge extends GraphComponent { Node from, to; }
```

If it is decided that every component of the graph should have an attribute describing its color, how can this be achieved? A natural first attempt could be to subclass the `GraphComponent` class, and add a `color` field to the subclass. This would, however, render the `Node` and `Edge` classes without any relationship to the new subclass, and thus still without a color attribute.

With multiple inheritance, we could create new subclasses of all the graph classes, e.g. as follows:

```
class GraphComponentX extends GraphComponent { Color color; }
class NodeX extends Node, GraphComponentX { }
class EdgeX extends Edge, GraphComponentX { }
```

Now, we have new class `GraphComponentX`, `NodeX` and `EdgeX` that all contain a `color` field. However, the `from` and `to` fields of `Edge` are still typed as `Node`, and this

necessitates a type cast if we are to access the `color` field through any of those fields, e.g.

```
EdgeX e = new EdgeX();
e.to = new NodeX();
((NodeX)(e.to)).color = Color.Red; // requires cast!
```

From this example, we see that *subclassing and refinement relationships are fundamentally different* [18], and that inheritance (even multiple) does not really help much when you want to refine classes in a class hierarchy.

Classboxes [18, 20] and Expanders [135] approach this problem by creating an explicit scope in which refinements to a class (or several classes) are valid. The latter provides a fully modular implementation as well. We want something similar, but with somewhat more flexibility, as stated in other desiderata, e.g. with regard to support for composition, renaming, and adaption.

This problem of refining classes in a class hierarchy can also, to a certain extent, be approached with virtual classes [83, 84]. If we defined the example above instead as an outer class with inner virtual classes, it could look something like this:

```
class Graph {
   virtual class GraphComponent { }
   virtual class Node extends GraphComponent { }
   virtual class Edge extends GraphComponent { Node from, to; }
}
```

In a language supporting virtual classes, we can now override the graph classes:

```
class GraphX extends Graph {
   override class GraphComponent { Color color; }
}
```

The new classes can be used as follows:

```
final GraphX g = new GraphX;
g.Edge e = new g.Edge();
e.to = new g.Node();
e.to.color = Color.Red; // no casts
```

There are now two versions of the graph library in the program, `Graph` and `GraphX` (and as such, the refinement is not performed in-place). This might be unproblematic (and even intentional and desirable), or it might be undesirable if the refined version should be used everywhere in the program.

Type safe application of virtual classes requires a somewhat sophisticated type system, and some language rules limiting the expressivity allowed [131]. Note for instance the `final` modifier in the example above (essentially making the `g` variable read only after its initial assignment), which is required for this to be type safe. In this thesis, we argue that a simpler approach can be taken, based on static composition without

runtime reification of the compositional hierarchy. In other words (and as indicated in Desideratum 6), we propose that a form of static multiple inheritance, where code is composed with a construct separate from those defining the type hierarchy, and where types are updated to reflect such compositions so that type casts can be avoided, can, in many cases, replace the need for ordinary multiple inheritance and virtual classes.

The discussion above is centered on classes, but the desideratum applies to both interfaces and generic constraints as well, for which refinement at any level in the hierarchy would be desirable.

Desideratum 10: ***Retroactive interface implementation.*** It should be possible to state that existing classes implement existing interfaces, without having to change the source code of either.

Discussion. Corresponding to what is possible e.g. with the `implementation` clause in JavaGI [136, 137], we want to allow a class that already matches an interface (i.e. it has a matching set of public signatures) to nominally implement this interface without resorting to source code changes. As such, the retroactive implementation of an interface can be seen as an in-place refinement, cf. Desideratum 9. It might often be necessary to *adapt* an existing class in order to make it conform to an existing interface specification (cf. desiderata 8, 9), so that a retroactive implementation clause can subsequently be specified.

As an alternative to retroactive nominal interface implementations, one could opt for structural interface conformance instead, such as in e.g. Whiteoak [56]. This provides a great deal of flexibility and opportunities for unplanned reuse. However, fully supporting structural subtyping has quite wide-reaching consequences both for the type system and for the nature of trust and blame-assignment for programs [101]. Thus, we argue that an optimal solution should support both structural and nominal conformance requirements.

4.3.3 Module Adaption and Composition in Dynamic Languages

As stated in Desideratum 1, our mechanism should be applicable to different object-oriented languages, even though the majority of our publications have focussed on Java-like languages. One class of languages within the object-oriented realm that in many ways differ significantly from Java, at least in terms of runtime semantics, consists of the *dynamic* languages, such as e.g. Groovy [116], Ruby [128], Python [103], etc. Such languages are typically dynamically typed, and often designated as “high-level” languages.

In this section, we discuss desiderata specifically for dynamic languages, where an overall desiderata is to provide a greater deal of flexibility and programmer control at runtime compared to the corresponding statically typed version of the mechanism.

Desideratum 11: ***Runtime composition and adaption.*** It should be possible to perform composition and adaption of program elements at runtime, based on runtime criteria.

Discussion. In dynamic languages, the programmer is typically equipped with quite a wide array of tools for adaption of the program at runtime, such as for instance runtime class modifications or the ability to directly make changes to live objects. Refinements in the form of runtime modification of classes can thus happen at any time, to any class (in general). This entails that any part of a program may perform a *global* state change and potentially affect wide-ranging parts of the application, even with seemingly innocuous changes to common classes.

In a dynamic language setting, we wish to support runtime composition and adaption of class hierarchies, while still making the feature relatively safe to use (compared to traditional runtime class modifications) by also supporting isolated adaption and composition, cf. Desideratum 7.

Desideratum 12: ***Meta-level control over composition.*** It should be possible for the programmer to control the composition of elements through a meta-level protocol.

Discussion. Dynamic programming languages typically support meta-level control over object-oriented semantics such as method call resolution and object creation through meta-object protocols or similar approaches. Correspondingly, as mentioned in the previous chapter, work has been done to provide such control also over AOP semantics. This desideratum entails giving the programmer the same level of control over composition with our mechanism, through a meta-level composition protocol. This includes providing control over composition semantics, such as e.g. conflict resolution, precedence, extended transformations etc.

4.3.4 Generic Programming and Adaption

As mentioned in Section 4.3.2, the ability to retroactively rename, adapt, and implement program elements also applies in the context of generic programming. This entails that an underlying assumption is that generic definitions are *first class*, and subject to the same possibilities as classes. Thus, Desideratum 8 also entails that it should be possible to adapt the generic constraints of a library through renaming. Correspondingly, Desideratum 9 also entails that it should be possible to perform in-place refinements of generic constraints, and Desideratum 6 also entails that it should be possible to *compose* previously unrelated generic constraints.

Also, semantics preservation (Desideratum 5) is important for generic code, and this entails that any constructs introduced for generic programming must support this property. That is, such constructs need to be complete in the sense that they facilitate separate semantic checking of the module in which they are contained, and any bindings to generic definitions must be preserved when generic units are composed.

Desideratum 13: ***Multi-type concepts and constraints.*** It should be possible to express non-trivial generic concepts in terms of a set of related classes, and it should correspondingly be possible to parameterize multiple classes with a single generic parameter.

Discussion. Just as for ordinary class libraries, we claim that (generic) concepts of some complexity typically span more than one type. Thus, one should be able to both express generic concepts that span more than one type, and correspondingly, to be able to parameterize more than one generic type with the same parameter, without repeating its constraints. This is important both for the sake of clarity and conceptual transparency, and also for the more pragmatic reason of minimizing repeated code.

One way to achieve this is to combine nested classes/interfaces and type parameters as in e.g. Scala [97] or JavaGI [136]. We believe, however, that using nested classes in this way results in a type system that, at least for many cases, is overly complex, and consequently somewhat difficult to understand and use. We thus intend to approach this in a simpler way, through template instantiation and compile-time specialization of generic definitions.

Desideratum 14: ***Propagation of concepts and constraints.*** It should be possible to specify generic concepts and constraints that propagate with module composition.

Discussion. Generics as implemented e.g. by Java and C# has a significant drawback in the fact that it is often necessary to needlessly (from a conceptual point of view) repeat formal type parameters and their constraints for subtypes of generic types or generic types that utilize other generic types [54]. The reason for this is that these languages do not support a mechanism for constraint propagation, as opposed to e.g. the suggested approach to Java generics by Thorup in [129].

With a mechanism that supports composition of modules, classes and interfaces will naturally "propagate" to the composed entity from the respective parts involved in the composition, and by supporting this also for generic parameters and their corresponding constraints we believe that heavily parameterized code can be written in a simpler and clearer manner.

Chapter 5

An Overview of the Package Template Mechanism

This chapter gives an overview of the Package Template (PT) mechanism as it is collectively described in the papers in Part II of this thesis. Note, however, that in some cases, our later research has replaced or subsumed the need for some of the constructs we present in earlier papers, and that this chapter presents the latest version of PT at the time of writing.

The PT mechanism is intended for object-oriented programming languages, and a package template is, in its essence, a collection of malleable/adaptable class definitions. Provided that the target language (e.g. Java, Groovy, etc.) supports them, other definition types such as interfaces and enumerations may also be included in package templates. The definitions inside a template may have interdependencies in the form of type hierarchies, signatures for methods and fields, variable types, generic constraints, etc.

The definitions within a package template cannot be referred to in a program before the template is *instantiated* in that program. Instantiation basically creates a fresh copy of the template declarations within the instantiating context (e.g. a package or another template), making each instantiation of a template independent of previous instantiations. This property lies at the core of the mechanism's abilities for adaption of the definitions within templates; since we can be sure that no-one else is using the definitions stemming from a given instantiation, we can safely perform several transformations in order to adapt these definitions to the problem at hand, without worrying about breaking code in other parts of the program.

In this thesis, we have worked with two distinct flavors of PT, one for statically typed object-oriented languages, and one for dynamically typed object-oriented languages.[1] In the following sections, we will give an overview of these two variants of PT.

[1] It bears mentioning, however, that features such as dynamic method invocation, reflection and even explicit localized dynamic typing in e.g. C# [89, sec. 4.7] muddies the distinction somewhat between dynamically and statically typed languages, as most languages have at least some features belonging in each paradigm. However, as a broad categorization this distinction will suffice for the purposes of this discussion.

Aside: a few words on syntax and terminology. It might be worth mentioning that some of the syntactical details (such as e.g. the `adds` keyword), and some pieces of the terminology (such as e.g. "explicit instantiations") in the following, are not what we would have chosen should we design the mechanism from scratch today. However, this thesis strives for a certain level of consistency between the current part and the published papers in Part II, and we will thus to a large extent use syntax and terminology from these papers, even though we through the clarity of hindsight sometimes might wish we would have chosen more wisely to begin with.

5.1 PT for Statically Typed Object-Oriented Languages

The development of PT for statically typed programming languages has used Java, arguably the most popular of the object-oriented programming languages in use today[2], as its underlying "host" language. However, a version for the Boo language has also been developed [118]. We will refer to the Java version of the package template mechanism as JPT.

5.1.1 Template Definitions

A package template in JPT looks much like a regular Java package, however, its contents are enclosed by a `template` construct with curly braces as delimiters. An example is shown below:

```
template T {
   class A { B b; ... }
   class B { ... }
   class C extends B { ... }
}
```

Inside the template `T`, there are three *template classes*, namely `A`, `B`, and `C`. Inside `A` there is a field declaration that is typed with `B`, and furthermore, `C` is a subclass of `B`. The contents of these classes are not shown here, but they can essentially contain the same kind of declarations and statements as ordinary Java classes can, with the addition of some PT-specific features and restrictions.

The syntax for template definitions is shown in Figure 5.1. Note that while the figure shows the full syntax, we do not treat every production in detail in this chapter; later chapters and the papers in Part II provide more details.

[2]See e.g. the TIOBE index, where Java in October 2012 is rated as the 2nd most popular language, below C: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, and the PYPL index, where Java in October 2012 was the most popular language regardless of paradigm: `https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language`

```
template-decl          ::=  template templ-name (? < templ-form-pars > ?)
                              (? expl-inst-clause ?) { template-body }
  templ-name           ::=  id (* . id *)

  // Formal template parameters, see Section 5.1.5:
  templ-form-pars      ::=  templ-form-par (* , templ-form-par *)
    templ-form-par     ::=  template id inst templ-name
                              (? < templ-def-act-pars > ?)

  // Explicit instantiation clause, see Section 5.1.5:
  expl-inst-clause     ::=  inst expl-inst-list
    expl-inst-list     ::=  expl-list-elem (* , expl-list-elem *)
      expl-list-elem   ::=  templ-name (? < templ-def-act-pars > ?)

  // Actual parameters supplied in an explicit instantiation of another template
  //    or to a formal template parameter:
  templ-def-act-pars   ::=  templ-def-act-par (* , templ-def-act-par *)
    templ-def-act-par  ::=  templ-name (? < templ-def-act-pars > ?)

  // Template body declarations:
  templ-body           ::=  (* templ-body-decl *)
    templ-body-decl    ::=  inst-stmt | templ-class-decl | templ-interface-decl |
                              adds-decl | required-type-decl
```

Figure 5.1: Syntax for template declarations. The *id* symbol is not explicitly defined, but follows corresponding Java rules for identifiers. The syntaxes for class and interface declarations in templates are not explicitly defined here, however they are roughly identical to their Java counterparts, with some notable exceptions detailed in the papers in Part II of this thesis, and the corresponding discussion for each paper in Chapter 6. The syntax for the *inst-stmt* clause can be found in Figure 5.2. The syntax for *adds-decl* is defined in Figure 5.3, while the syntax for required types is defined in Figure 13.1 in Chapter 13.

Legend: Non-terminal symbols are written with a regular font, while terminal symbols are written with a **bold monospace** font. Optional clauses are delimited by parenthesis with question marks, while zero or more repetitions of one or more clauses is denoted by parenthesis with asterisks.

```
inst-stmt                ::=  (? id : ?) inst templ-name (? < templ-act-pars > ?)
                                (? with-spec ?) ;

  // Actual parameters supplied in the instantiation of a parameterized template:
  templ-act-pars         ::=  templ-act-par (* , templ-act-par *)
    templ-act-par        ::=  templ-name (? < templ-act-pars > ?) (? with-spec ?)

  // A with specification can contain renames and concretizations:
  with-spec              ::=  with single-with-spec (* , single-with-spec *)
    single-with-spec     ::=  type-rename | req-type-concr

      // Renaming of types and attributes:
      type-rename        ::=  id => id (? attr-rename-clause ?)
                                (? nested-rename-clause ?)
        attr-rename-clause ::=  ( attr-rename (* , attr-rename *) )
          attr-rename    ::=  id (? method-pars ?) -> id
            method-pars  ::=  ( id-list )
              id-list    ::=  id (* , id *)

        // Renaming of nested types:
        nested-rename-clause ::=  { type-rename (* , type-rename *) }

      // Concretization of a required type:
      req-type-concr     ::=  id (? attr-rename-clause ?) <= id
```

Figure 5.2: Syntax for `inst` statements. The *id* symbol is not explicitly defined, but follows corresponding Java rules for identifiers. The *templ-name* symbol is defined in Figure 5.1.

The legend for the syntax definitions can be found in the caption for Figure 5.1.

5.1.2 Instantiations

A template is instantiated with the `inst` statement, the syntax of which is shown in Figure 5.2. For instance, the template `T`, as shown in Section 5.1.1 above, could be instantiated in a package `P`, as shown in the second line of the example below. For packages, we will in this thesis use a syntax where curly braces enclose the package contents, corresponding to how templates are declared, in order to make the package boundaries explicit to the reader.

```
package P {
   inst T with A => AA, B => BB;
   // class C will be implicitly imported into P
   class D { ... }
}
```

The `inst` statement in package `P` above makes the template classes `A`, `B`, and `C` available in that package. Note that the classes `A` and `B` will be known under the new names `AA` and `BB`, respectively, in `P`. This is specified by the rename clause of the `inst` statement. The renaming is "deep", meaning that it is performed based on the semantic bindings established in `T`.

This entails that the variable `b` in the class `AA` will now be of type `BB`, and also, that the class `C` will extend `BB` in `P`. The class `D` is defined directly in `P`, and is an ordinary Java class. The class names from the template `T` (here `A` and `B`), cannot be referred to in `P`, except in some very specific cases that have to do with disambiguation.

Attributes of classes (and interfaces etc.) may also be renamed. Renaming of an attribute must be done on the class (or interface etc.) where the attribute is originally defined. That is, you cannot for instance rename a method in a class if that method is an override of a corresponding method in a superclass.

For renaming attributes, a different arrow is used (`->`). If, for instance, we wanted to rename the variable `b` in the example template `T` above to `bb`, the `inst` statement would read

```
inst T with A => AA (b -> bb), B => BB;
```

The example package `P` above shows how a template is instantiated in a package. However, templates may also be instantiated in other templates, and one template or package can even instantiate another template multiple times. Cyclic instantiations of templates instantiating templates are forbidden.

The `inst` statement results in a template instance being created at compile-time, and this instance can be named, as shown in the first production of the syntax overview in Figure 5.2. For example, the instance of `T` in the package `P` can be named `X` by writing the `inst` statement as follows:

```
X: inst T with A => AA, B => BB;
```

Named instances are only used for the purpose of disambiguation, e.g. when the same template is instantiated twice, and there is a need to differentiate between the respective instances. This typically only occurs in constructor implementations, since PT rules prevent ambigous names in most other situations.

```
adds-decl          ::=  adds-class | adds-interface | adds-req-type

  adds-class       ::=  class id adds (? extends-clause ?) (? implements-clause ?)
                          { adds-class-body }

  adds-interface   ::=  interface id adds (? extends-clause ?)
                          { adds-interface-body }

  adds-req-type    ::=  required ( r-type-adds | r-class-adds | r-int-adds )
    r-type-adds    ::=  type id adds (? extends-clause ?) { adds-rtype-body }
    r-class-adds   ::=  class id adds (? extends-clause ?) (? implements-clause ?)
                          { adds-rclass-body }
    r-int-adds     ::=  interface id adds (? extends-clause ?) { adds-rint-body }
```

Figure 5.3: Syntax for `adds` clauses. The *id* symbol is not explicitly defined, but follows corresponding Java rules for identifiers. The syntax for the bodies of the additions are not explicitly defined above, but for addition classes and addition interfaces they roughly follow that of ordinary Java classes and interfaces. The body of an addition for a required type follows the same syntactical rules as for ordinary required types, which are treated in more detail in Paper VI in Chapter 13. The syntax for the *extends-clause* and *implements-clause* productions are equal to their corresponding Java counterparts.

The legend for the syntax definitions can be found in the caption for Figure 5.1.

5.1.3 Additions

The classes (and interfaces, etc.) of a template can be refined through *additons*. Additions to template classes are defined as class-like constructs called *addition classes*. Addition classes can contain new declarations, as well as overrides for methods defined in corresponding template classes. Below, we see an example of what this might look like for an instantiation of the template `T`, as defined in Section 5.1.1 above:

```
package P {
   inst T with A => AA, B => BB;
   class AA adds {
      ... additional attributes and overrides in AA ...
   }
   class BB adds {
      public int i = 41;
      ... additional attributes and overrides in BB ...
   }
```

```
    // class C will be implicitly imported into P,
    // and will be a subclass of BB

    class D { ... }
}
```

The contents of an instantiated template class and the contents of the corresponding addition class will be concatenated in `P`, and it will thus be safe to access e.g. the newly added field `i` through the existing field `b` in the class `AA`. For instance, a method in the addition class `AA` can safely contain a statement such as

```
this.b.i++;
```

even though the field `b` was originally typed with `B`, which does not contain any variables named `i`. Note that no type casts are required.

In an addition class the programmer may override a method from a corresponding template class. The original overridden method can be called by prefixing the call with the `tsuper` (for "template super") keyword, e.g. `tsuper.m()`. More detail on overrides in subclasses and addition classes, and how lookup is performed in each case, can be found in Paper V in Chapter 12.

5.1.4 Merging

In PT, previously unrelated classes can be merged. Syntactically, this is achieved by using the renaming capabilities of the `inst` statement and renaming two or more classes *to the same name*. The new class is created by taking the disjoint union of all the attributes of the instantiated classes (with potential renames performed), together with the attributes of the common addition class. We use a disjoint union in order to separate equal signatures stemming from different (instantiations of) template classes. An implementation must present an error message if there are equal (for methods, override-equivalent [58, sec. 8.4.2]) signatures stemming from two or more classes in the resulting merged class.

Consider the simple example below:

```
template T {
  class A { int i; A m1(A a) { ... } }
}

template U {
  abstract class B { int j; abstract B m2(B b); }
}
```

Consider now the following usage of these templates:

```
package P {
   inst T with A => MergeAB;
   inst U with B => MergeAB;
   class MergeAB adds {
      int k;
      MergeAB m2(MergeAB ab) { return ab.m1(this); }
   }
}
```

These instantiations result in one single class named `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. The order of the instantiations is insignificant (i.e., it does not matter if `T` or `U` is instantiated first in the example above).

Note how the abstract method `m2` from `B` is implemented in the `adds` clause, and furthermore how both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`. The resulting class `MergeAB` is no longer abstract.

Issues with regard to class merging, such as name clashes, conflict resolution, addition classes, virtual methods, constructors, etc., are treated in more detail in Paper V in Chapter 12.

5.1.5 Parameterization

In PT as defined by this thesis, there are two options for parameterization of templates. The first is through *required type specifications*, as defined in Paper VI in Chapter 13, and the second is through the use of templates as parameters to other templates, as outlined in Paper V in Chapter 12, and further discussed in Section 6.5.2.

Required type definitions. A required type is a template-wide abstract type definition, on which other declarations in a template might depend. The abstract declaration must be concretized at the latest when a template is instantiated in a package. Utilizing required types, the programmer can specify both nominal and structural constraints for the same type. A required type declaration is declared within a template at the same level as ordinary classes, e.g. as follows:

```
template T {
   required type R implements I { void m(); }

   class A {
      void m(R r) { ... r.m(); ... }
   }

   class B {
      R r;
      ...
} }
```

The required type `R` in the example above has a nominal constraint to implement an interface `I` (presumably declared in an imported package), and furthermore a structural constraint to implement the `void m()` method. The classes `A` and `B` are both parameterized by `R`.

Upon instantiation of `T`, the required types may be concretized by utilizing the `<=` arrow, as shown in the production *req-type-concr* in the syntax overview for the `inst` statement in Figure 5.2. Attributes of required types may be renamed to match the supplied parameters, as shown below for the method `m`:

```
package P {
   // We concretize the required type R with C.
   // The structural requirement for a method m()
   // is changed to match the method f() in C.
   inst T with R (m() -> f) <= C

   class C implements I {
      void f() { ... }
      ...
   }
}
```

Furthermore, required types may be merged together, in order to e.g. combine requirements from multiple templates.

Required types may also be utilized in combination with ordinary Java-style generics, for instance by supporting parameterization with generic Java types as bounds. That is, both required types, and ordinary template classes and interfaces, may have conventional type parameters.

Template parameters to templates. In addition to using required types for parameterization of templates, a package template may also be parameterized through *template* parameters, using other templates as bounds. This allows for a looser coupling between templates that instantiate templates and templates that are instantiated by other templates. This also allows for combining different extensions to a common template, which turns out to be very useful in order to obtain flexible solutions to the expression problem [134] and similar problems.

Given the template `T` as defined above on Page 48, other templates can be parameterized using `T` as a bound, e.g. as shown for the template `U` below:

```
template U<template X inst T> {
   // utilize definitions from T ...
}
```

In this example, the name `X` is used as a formal parameter name. For a template to be useable as an actual parameter for a template with a formal parameter with bound `T`, its declaration must include an *explicit* instantiation of `T`, as shown in the *explicit-inst-clause* production of Figure 5.1, and in the example below:

```
template T2 inst T { // T2 has an "explicit inst" of T
   // Refine definitions from T and/or introduce new
   // definitions here. Names stemming from T cannot
   // be renamed.
}
```

The template `U` can now be instantiated by supplying either `T` or `T2` as an actual parameter for `X`.

5.1.6 Aspect-Oriented Programming Constructs

In Paper I (Chapter 8), we discuss an extension of PT where a template class can contain definitions of the aspect-oriented programming (AOP) [51] constructs pointcut and advice. The pointcut definitions allowed in this variant of PT resemble those of AspectJ [33], but they are quite severely limited with respect to the joint points over which they may range. In short, the pointcuts of PT are limited to only refer to definitions within the classes in which they themselves are defined. This might seem like a crippling restriction for AOP programming, and in many ways it is, for the traditional sense of AOP. However, it also provides the programmer with compile-time safety guarantees for pointcuts. Furthermore, it turns out that even in this very limited form, pointcuts are still quite useful in order to signify that behavior (in the form of an advice) should occur before/after/around the execution of a certain set of, typically, method calls. This becomes especially useful when abstract pointcuts are utilized in template classes, in order to make explicit the contract that something to which the class will react will have to be explicitly concretized by the programmer using the template.

5.2 PT for Dynamically Typed Object-Oriented Languages

The work on a version of the package template mechanism for dynamically typed languages has in this thesis been centered around the Groovy programming language. Using Groovy as the basis for a prototype implementation is beneficial for several reasons, including the language's strong support for developing embedded domain specific languages (DSLs), paired with its capabilities for meta-level programming. Furthermore, as a basis of comparison with the statically typed variant of the mechanism, Groovy is well-suited because it is in many respects very similar to Java, both in terms of syntax and semantics, with the important distinction that bindings in Groovy, of course, are resolved primarily at runtime. The Groovy implementation of PT is called GPT for short.

5.2.1 Template Definitions

A template definition in GPT closely resembles the corresponding definition in JPT, only with Groovy classes (and interfaces, etc.) instead of their Java counterpart. More

details on this can be found in papers III and IV in chapters 10 and 11, respectively, but we present a short overview in the following.

5.2.2 Instantiations

Instantiation of templates in GPT is performed during program execution, as opposed to being a compile-time statement in JPT. An instantiation is described by a set of calls to methods that collectively make up a small internal DSL for template instantiation, accessed through a method `instantiate` defined in the supplied `PT` class.

The big advantage to using an internal DSL (besides not requiring a separate parser/compiler/etc), is that the DSL can be *extended and refined by the user*, and this is utilized heavily in Paper IV. Below is an example of instantiating a template `T` and renaming the classes `A` and `B` to `AA` and `BB`, respectively. (Note that the template `T` is not *defined* here, the code below just shows an instantiation of it. The template is presumed to be defined elsewhere, with the classes `A` and `B`.)

```
def INST_T = PT.instantiate {
   template T {
      map A, AA
      map B, BB
   }
}
```

The *variable* `INST_T` will thus, after this is executed, hold an instance of `T`, with classes `AA` and `BB`. The variable can be reassigned multiple times throughout the execution of the program, and can thus hold references to other instances, or to completely unrelated data such as strings and integers.

Objects can be made from the classes in the instance through the variable, e.g.:

```
def aa = new INST_T.AA(...);
```

5.2.3 Addition classes

Addition classes in GPT are declared as ordinary classes with a `@PTAddition` attribute within the same file as the instantiation. Which addition class to use for an instantiated template class is resolved at runtime (even though the addition class itself, like ordinary classes, is declared statically in Groovy), and thus a single addition class can be an addition for multiple, independent instantiations of template classes (in contrast to the statically typed version of the PT mechanism).

Below is a small but complete example of GPT program with a template, a main class that instantiates this template, and an addition class that overrides a method from a template class:

```
template T { class A { def f() { /* some code here */ } } }

class Program {
  def INST_T = PT.instantiate {
     template T {
        map A, AA
        map B, BB
} } }

@PTAddition class AA { def f() { /* some other code here */} }
```

5.2.4 Strategies

For the statically typed versions of PT that we have considered, the semantics of an instantiation is defined once, and cannot be modified by the programmer. For the dynamic version, on the other hand, the programmer may adapt the instantiation semantics to the problem at hand, and even have several different semantics in one single program, that may be plugged in or out based on runtime conditions. The programmer may achieve this by supplying *strategies*, which are classes that implement certain interfaces, in the call to the `PT.instantiate` method. Such strategies may e.g. provide alternate semantics for conflict resolution, resolve precedence issues, or modify the specification of what is to be instantiated, and how, in interesting ways.

Strategies are supplied as part of the instantiation specification parameter to the `instantiate` method, e.g:

```
def INST_T = PT.instantiate {
   strategy S {
      template T {
         ...
} } }
```

In the example above, `S` is the name of a class, defined by the programmer, that implements a strategy interface.

In Paper IV we utilize strategies to, amongst other things, provide conflict resolution semantics resembling that of JavaFX [42, 50], and to add AOP-like constructs to the instantiation specification.

5.3 PT in This Thesis Compared to Previous Publications

As mentioned, the PT mechanism was proposed in [76] before work on this thesis was started. While the rest of this chapter has tried to present the mechanism as a coherent whole, we will in this section list what was there before the work on this thesis started, and what has been added specifically through the work with this thesis.

The basic constructs of the mechanism were already described. This includes:

- The basic idea, including the definition and instantiation of templates with class hierarchies.
- Class merging and renaming of classes, methods and fields.
- Refinement through addition classes.
- Parameterization with simple, Java-like, generic parameters in the template header.

Specific contributions to the PT mechanism itself from the work presented in this thesis include:

- A semantics for the core concepts of the mechanism defined through a translation to Java. This includes the basic constructs listed above.
- A set of simple AOP extensions that allows the definition of pointcut and advice in template classes.
- The `tsuper` construct, which facilitates calling template class methods that are overridden in an addition class, and accessing field definitions that are shadowed by definitions in an addition class.
- The `tabstract` modifier, which designates abstract definitions that must be concretized in an addition class (even `static tabstract` definitions are possible).
- A scheme for how constructors can be handled in the face of addition classes and class merging.
- The `extends external` construct, which allows template classes to be subclasses of existing classes outside of templates.
- An extension of the mechanism that allows templates to be parameterized using other templates as bounds, and correspondingly that templates may be supplied as actual parameters in the instantiation of a template.
- Required types as first-class, propagating, type parameters which can be refined, adapted, and merged.
- Dynamically typed PT in its entirety, including runtime instantiations and a protocol for meta-level control over such instantiations.
- A compiler for the Java version of the language, developed in cooperation with other researchers and students in the SWAT project.

Furthermore, the overall contributions of this thesis are not only related to new constructs in the PT mechanism per se, but also includes examples, implementations, descriptions, discussions, etc. The basic constructs of the mechanism, as shown in the first bullet point list above, have thus also benefited from this thesis work, in the sense

that they have been solidified, tested, refined, and more clearly defined. The work on this thesis has thus revolved around the PT mechanism as a whole, and not only been focussed on adding new constructs to the mechanism.

Chapter 6

Overview of Research Papers and Discussion

In this chapter, we briefly recount the main ideas and contributions of each of the papers that are included in their entirety in Part II of this thesis, and provide a short discussion for each paper. The papers are presented chronologically according to their publication dates (which do not always correspond with the order in which they were written, mainly due to variations in publication time).

Each paper is discussed in its own section, which starts with a short description of the nature of each author's contribution to the work. For venues with published acceptance rates, this is listed along with the general publication information.

Following the author information, there is a summary section, where we try to present enough of the subject matter of the paper so that the discussion can be read without having intimate knowledge of the paper to which it refers, however, we have also aimed to keep the summary reasonably short, since, after all, paraphrasing the entire paper here does not make much sense. Thus, it might be beneficial to read through the actual paper to which the discussion pertains first.

The discussion for each paper aims to view that paper in light of the rest of this thesis, and to address insights gained since the paper was written, and also, in some cases, include material that we could not find room for in the paper due to publication venue limitations. Furthermore, for some of the papers we discuss important related work that has been published after the respective papers' publication dates.

For some of the papers, certain parts of the discussion are quite technical in nature, and build more or less directly on the material presented in the paper. For these sections, we will try to be explicit about this, and refer to the paper in question for more background.

A guide to reading the papers. As mentioned, the papers can be read independently, however, there are arguments for reading the papers in a different order than the chronological one in which they appear in Part II of this thesis.

A good starting point when reading the papers might actually be Paper V, entitled *Challenges in the Design of the Package Template Mechanism*, which can be found in

Chapter 12. This paper presents a pretty good overview of the Package Template mechanism, and discusses many of the fundamental issues that we have had to deal with when designing the mechanism. It should as such provide a decent background for understanding the remaining papers.

Papers I and II, in chapters 8 and 9, respectively, deal mainly with *applying* the mechanism to the task of creating reusable design patterns. Furthermore, Paper I also introduces a limited version of AOP constructs to the PT mechanism, that respects the boundaries of traditional object-oriented encapsulation principles. Paper II contains a piece of the material included also in Paper I; in particular, Section 9.3.4 is more or less directly lifted from Paper I. Thus, papers I and II are best read in sequence.

Papers III and IV, in chapters 10 and 11, respectively, represent a slight detour in this thesis, in the sense that they approach the package template mechanism from the context of dynamically typed languages, and the issues at hand are thus mostly of a somewhat different nature than for the rest of the papers. Papers III and IV should be read in sequence, as Paper IV builds directly on the constructs introduced in Paper III, but other than that they can be read either before or after the preceding two papers.

In Chapter 13, we find Paper VI. This paper presents an extension to the statically typed variant of PT that adds new constructs for generic programming. In that sense, it builds on Paper V, but it is otherwise independent from the rest of the papers.

The final paper in this thesis is Paper VII, in Chapter 14. This paper defines the semantics of (a core subset of) the package template mechanism for Java through a transformation that, given a legal PT program, will produce legal Java code from PT code. It can be read independently of the other papers, but it would be a good idea to have read Paper V first.

6.1 Paper I: *A Reusable Observer Pattern Implementation Using Package Templates*

Authors. Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. Axelsen is the main author, and was responsible for the initial idea for this application of PT, and did almost all of the writing of the paper. Sørensen and Krogdahl provided important feedback and helped shape both the idea and the paper.

Publication. Published in Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2009 [11]. The workshop was part of the AOSD 2009 conference.

6.1.1 Summary

In this paper, we explore a potential addition to the package template mechanism in the form of a limited set of aspect-oriented programming (AOP) capabilities, based on a subset of the pointcut and advice constructs of AspectJ [5, 33]. The AOP constructs differ in scope and expressive power from those of AspectJ; to begin with, aspects are

expressed not as separate entities (e.g. class-like constructs as in AspectJ), but rather, pointcut and advice declarations are defined as attributes of template classes, corresponding to how ordinary methods and fields are defined. Specifically, this means that the join points visible to any given pointcut all reside within the same class as the pointcut itself, or a superclass of this class. This is in contrast to the global view that aspects enjoy in e.g. AspectJ.

Since pointcut and advice declarations are defined within classes, they may be refined or redefined in subclasses or in addition classes, in much the same way as methods.

The constructs proposed in the paper are applied to the Observer design pattern [53]. The example is small and relatively trivial, from a conceptual standpoint. Still, it is vital for many real-world applications, and creating a pluggable, directly reusable, and type safe implementation is not a trivial task in mainstream object-oriented languages such as Java.

The paper shows how such an implementation can be made with PT, and exemplifies how this implementation can be applied to previously unrelated code through instantiations with class merges, and refinement of abstract pointcuts.

6.1.2 Discussion

The Observer pattern example presented in the paper is interesting not only because it has been studied and used as an example many times over in the literature (e.g. [65, 87, 114, 126]), but also because it represents a rather fundamental behavior for computer programs: to observe and then react when *something of interest* happens. The abstraction of this pattern, without having to prematurely specify what this something is, is at the heart of creating a reusable implementation of the pattern.

This is also the basis for a lot of common utilizations of aspect-oriented programming, including the ever-popular example of logging the dynamic behavior of a program under certain conditions.

In the following sections, we will consider the interaction of the small set of AOP features presented in Paper I with the `tsuper` and `tabstract` constructs, which are introduced in Paper V.

Furthermore, we did not have room in Paper I to discuss the semantics of the mechanism, barring a few examples, and we will thus describe the semantics of the AOP constructs in PT here, in terms of a translation to corresponding AspectJ [5, 33] aspects. (In Paper VII we define semantics of PT itself, without any AOP constructs, in terms of a translation to plain Java.)

We will also briefly discuss whether it is indeed a good idea to include both `call` and `execution` join points in PT, and compare our approach with AOP to that of event-driven programming.

Abstract declarations and overridden template class methods. The example studied in this paper utilizes abstract classes and abstract class attributes (methods and

```
// Pointcut declarations:
pc-decl      ::=  (? modifier ?) pointcut ( qualified-identifier | void | * )
                    identifier ( (? parameter-list ?) ) ( { pc-expr } | ; )

  // Pointcut expressions:
  pc-expr   ::=  ( call | execution | get | set ) ( identifier ) |
                 pc-expr && pc-expr |
                 pc-expr || pc-expr |
                 ( pc-expr )
```

Figure 6.1: Syntax for pointcuts in PT, as defined in Paper I. The **&&** (and) operator takes precedence over the **||** (or) operator, as for ordinary boolean expressions in Java. The legend for the syntax definitions can be found in the caption for Figure 5.1.

pointcuts) to specify that certain parts need to be concretized by subsequent addition classes.

In Section 12.3.4 of Paper V, we introduce the notion of a `tabstract` modifier, which clarifies the distinction between things that *can* be concretized by a sublcass or an addition class (`abstract`), and things that *must* be concretized by an addition class (`tabstract`). In the examples in the paper, all `abstract` attributes would have been labeled as `tabstract` in the current version of PT.

Furthermore, in Section 12.3.2 of Paper V we introduce the `tsuper` construct, which the programmer can use to call template class methods that are overridden in an addition class. If we had had the `tsuper` construct available at the time of writing for Paper I, it would have been natural to allow pointcuts in addition classes to refer to pointcuts in template classes through `tsuper`, so that refinements could be done without repeating the original pointcut.

Semantics of the AOP constructs in terms of AspectJ. The semantics of the AOP constructs are not described in detail in Paper I. Below, we sketch the semantics in terms of their translation to the AspectJ language [5, 33]. As such, the following paragraphs are somewhat technical in nature, and experience with programming in AspectJ would probably be beneficial in order to fully comprehend the description of this translation.

As mentioned in the paper, pointcuts in PT are local to their defining class, including its subclasses and addition classes. Pointcuts may as such only refer to attributes defined within their enclosing class or any of its superclasses, following the OO principle of encapsulation. Thus, it is invalid in a pointcut to refer to a definition that is not directly accessible from within the defining class. We recall the syntax for pointcuts in PT from Paper I, shown here in Figure 6.1 in the same style as the general syntax overview for PT from Chapter 5.

A PT-style template class with pointcuts and advice can be transformed into a pair of an ordinary class and an AspectJ-style aspect by following the syntax-driven algorithm-sketch below. The algorithm can be applied when a template is instantiated

in a package, and all additions, merges, etc have been handled, and all conflicts thus are resolved.

- For each template class with aspect-oriented definitions, we create a corresponding AspectJ-aspect. The aspect needs to have the `privileged` modifier, since advice and pointcut declarations might refer to private declarations within the template class.

- For each pointcut P_i in the template class, a corresponding AspectJ pointcut P'_i is created in the new aspect. Each reference to an attribute (method or field) in P_i is in P'_i prefixed with an explicit reference to the class in which P_i was defined, for method references including a + sign to indicate that overrides in subclasses are taken into account as well. That is, if the pointcut expression of P_i for instance was defined as `call(m())` in a template class `C`, the AspectJ counterpart is `call(<returntype> C+.m())`.

- Since advice in the template class typically will refer to fields and methods in that class, the pointcut will need to have a `target(c)` specification, which can be used in the AspectJ-style advice to replace (explicit or implicit) `this` references from the PT advice. The target will consequently also need to be listed in the AspectJ-style pointcut's formal argument list, followed by any other formal arguments from the PT-style pointcut. Thus, for instance, if the pointcut declaration in a template class `C` was e.g.

  ```
  pointcut void <pc-name>(P1 p1, P2 p2) { call(f) || call(g) }
  ```

  where `f` and `g` are methods that both return void and both have two formal arguments of type `P1` and `P2`, the corresponding AspectJ-pointcut would be:

  ```
  pointcut <pc-name>(C c, P1 p1, P2 p2) :
     ( call(void C+.m(P1, P2)) || call(void C+.f(P2, P2)) ) &&
     target(c) && args(p1, p2);
  ```

- For each advice in the template class, a corresponding AspectJ-style advice must be created, where each reference to a method or variable defined in the template class is prefixed with a reference to the target in the AspectJ advice. (If the reference is prefixed with the `this` keyword, that keyword will be *replaced* by the target reference.)

We see that pointcuts and advice as described in Paper I with relative ease can be transformed into a corresponding AspectJ-implementation. The main point of the PT variant is however not the syntax, but the restrictions with regard to encapsulation that make the AOP features safer to use (and, obviously, limits their expressiveness), paired with PT's abilities for retroactive refinement and composition.

Call versus execution join points. As can be seen from the syntax for AOP constructs in Figure 6.1, the pointcuts in this extension of PT support the specification of both `call` and `execution` join points (as defined by AspectJ). The difference between these two constructs is primarily that the former matches the call site, while the latter matches the execution of the method body. Thus, the former will require weaving at the call site, while the latter can be achieved by modifying the target of the call.

Since a main goal of Paper I is to add AOP constructs to PT in a way that does not have wide-ranging consequences for existing code, it can be argued that the `call` join point specifier does not fulfill this goal, since it requires changes to code (in compiled, byte code, form) outside the template, and that this option thus should be removed. (All the examples would work equally well with `execution`.)

However, also pointcuts containing the `execution` specifier might require changes to code outside of the template, for instance if a template class is subclassed by a package class, or if package classes implement (instantiated) template interfaces. Still, this might be deemed more acceptable since subclasses of template classes and implementors of template interfaces are more closely coupled to each other. Of course, an alternative would be to define the semantics of the PT AOP constructs so that they *only* affect template-code, and not external subclasses or interface implementors.

AOP in PT versus programming with events. For languages without AOP support, the Observer pattern is typically implemented using *events*. Event-driven programming is a programming style where the programmer defines *event handlers* that are triggered by events that occur during the execution of the program. Events can fire due to external happenings such as e.g. a mouse click, or they can be fired explicitly by the program itself, for instance when a certain condition is met. Programs can be written in an event-driven style in most languages, but some languages provide high-level constructs specifically tailored for such programming.

The notion of a join point in AOP is in many ways similar to that of an event, in the sense that it represents a location in the program that the programmer may "hook onto" and perform some action. Unlike an event, however, a join point is not explicitly specified as such by the programmer. Rather, it is based on the underlying semantic model of the language, and can for instance be a method call, a field access, etc.

An advice represents the "event handler" in AOP. Typically, an advice may cancel the program's original behavior (e.g. omit the call to `proceed(...)` in AspectJ), while an event handler may only *react* to events, unless the programmer has made explicit support for overriding the default behavior (e.g. by setting the `Handled` property of the event's argument to `true`, as typically supported by key press event handlers in the .NET framework[1] and similar implementations).

Event subscription (or listening) is typically restricted by the same visibility rules as ordinary method calls in object-oriented programming (i.e. it adheres to standard scoping and access rules), in contrast to the typical approach of AOP that allows pointcuts to range over the entire program. Event-based programming is thus, in that sense,

[1] See for instance `http://msdn.microsoft.com/en-us/library/system.windows.input.keyeventargs(v=vs.110).aspx`

closely related to our approach in Paper I.

Several programming languages come with explicit language support for defining and firing events, making the implementation of subjects and observers more convenient. An example of such a language is C#, which has built-in support for the so-called *delegate event model*. Given a PT implementation for that language, a reusable, pluggable, observer pattern implementation could e.g. be written as follows:

```
template ObserverProtocol {
   class Observer {
       tabstract void Notify(Subject changee);
   }
   public class Subject {
      delegate void ChangedEventHandler(Subject sender,
         EventArgs e);
      event ChangedEventHandler Changed;
      void OnChanged() {
         if(Changed != null)
            Changed(this, new EventArgs()); // notify observers
} } }
```

The `ChangedEventHandler` in the example above declares a signature for *delegates* (or function pointers) that can be used as listeners (event handlers) for the event `Changed`. The `OnChanged` method triggers (fires) the event if there are any listeners attached to it.

Corresponding to the example from Section 8.3.1 in Paper I, a package for displaying objects on a screen with automatic update of the screens when interesting lines change could be written as shown below. The `Drawing` template is the same as the one presented in Paper I.

```
package DrawingPackage {
   inst Drawing with Line => LineSubject,
      Screen => ScreenObserver;
   inst ObserverProtocol with Subject => LineSubject,
      Observer => ScreenObserver;

   class LineSubject adds {
      void setStart(int x) {
         tsuper.setStart(x);
         OnChanged();
      }
      // corresponding implementation for setEnd here
   }

   class ScreenObserver adds {
      void Notify(Line l) { ... }
} }
```

We utilize the `tabstract` and `tsuper` constructs in this example. As mentioned above, the `tabstract` construct was not part of the PT design when Paper I was published, and the same applies to `tsuper`.

As can be seen from the example, the use of events, in languages that provide adequate support, can subsume *some* of the need for AOP, and in this example the implementation with events is relatively straight-forward. However, note that the implementation with events is not able to capture the *explicit* requirement of an abstract pointcut that, upon instantiation, a set of interesting join points *must* be specified by the programmer. This point is indeed one of the main arguments for keeping pointcut and advice constructs in PT even for languages that support explicit event declarations.

The EScala language [55] takes the conceptual overlap between events and join points one step further, and extends the Scala language [96] with imperative C# -style events and declarative implicit events, as previously mentioned. The implicit events are analogous to the join points of AOP, allowing events to be triggered for method calls. Corresponding to our implementation of pointcuts in PT, the EScala event declarations follow the encapsulation rules of the host language; that is, the EScala mechanism follows the essential encapsulation rules of Scala, as PT follows essential the encapsulation rules for Java. EScala also supports event composition, allowing the expression of higher-level events that are expressed as the union or the intersection of other events.

6.2 Paper II: *Towards Pluggable Design Patterns Utilizing Package Templates*

Authors. Eyvind W. Axelsen and Stein Krogdahl. Axelsen is the main author, and was responsible for the initial idea for this application of PT, and did all of the writing of the paper. Krogdahl provided important feedback and helped shape both the idea and the paper. The paper incorporates a part that includes previously published material from Paper I, for which also Sørensen provided feedback.

Publication. Published in Proceedings of Norsk Informatikkonferanse (NIK, Norwegian Informatics Conference) 2009 [7].

6.2.1 Summary

Paper II takes the idea from Paper I, and builds on and extends this work; Paper II provides examples of applying PT to provide *pluggable* variants of the *Singleton* and the *Memento* design patterns in addition to the Observer pattern, which was treated in Paper I. All three design patterns were first described in the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et al. [53].

The Memento pattern, like the Observer pattern, is a behavioral pattern. It describes how an object may represent its internal state (both current and former) as separate objects, and utilize this to support rollback/undo operations.

The Singleton pattern is a creational pattern, the purpose of which is to limit the number of objects of a class to maximum one. The pattern prescribes a single method that transparently creates an object of the class the first time it is needed by the application, and returns the same object upon subsequent requests. While the pattern is conceptually very simple, the intricacies of the memory model of Java makes it, perhaps surprisingly, non-trivial to manually implement correctly in a thread-safe manner; a correct implementation depends not only on the source code itself, but also the version of the Java runtime environment on which it is supposed to run [13].

6.2.2 Discussion

The main aim of Paper II is to present examples of how the PT mechanism can be applied to create *reusable* implementations of design patterns, an area that has been identified as problematic by Soukup [113] and others. In addition to that, Paper II also contains brief discussions on access modifiers and nested classes in relation to the pattern implementations.

Below, we take a new look at some technical issues that are related to the PT mechanism and the examples presented in the paper. Some issues are treated in more detail in later papers or discussions, while others are treated directly in this section.

For completeness, we will here consider all the remaining design patterns from [53] (Paper II only presents three patterns), and how their respective implementations in PT differ from their Java and AspectJ counterparts. But first, we will, as mentioned, discuss some technical issues with regard to some particular constructs in PT, and interactions with PT features that are introduced in subsequent papers.

Constructors. The PT implementations in Paper II of the Singleton and Memento patterns make use of constructors in template classes. We have, in subsequent papers, and in the implementation of PT for Java, refined the way constructors work in PT. In particular, see Section 12.3.5 of Paper V on constructors, and the corresponding discussion for Paper V in Section 6.5.2.

Specifically, the constructors within the classes in the pattern implementations would have to be accompanied by a corresponding `assumed` constructor declaration, in order to conform to the latest implementation of PT for Java. An `assumed` constructor declaration is a promise that a certain constructor signature will be available when the template is implemented in a package, and it thus puts a requirement on the package class programmer to implement the specified constructors. Thus, `assumed` constructor declarations are required if there are `new` statements in template class code that creates objects of template classes.

Access modifiers. There is a brief discussion on access modifiers in Section 9.3.2 of Paper II, but space constraints prevented us from a more thorough treatment. We get back to this issue in Paper V, in Section 12.3.8. Note the discussion on the `protected` modifier in that paper, and how it expands on the discussion in Paper II. Still, access modifiers and how to best incorporate them in PT remains pretty much an open issue.

Abstract declarations. The examples in Paper II, as those in Paper I, utilizes abstract attributes (in the form of methods and pointcuts) to specify that certain parts need to be concretized by subsequent addition classes. Like for Paper I, it would in the current version of PT be better to declare these attributes with the `tabstract` modifier, as introduced in Paper V.

Nested template classes. Paper II only very briefly mentions nested classes, and they are not treated in any detail in the rest of the papers in this thesis. However, for many Java programs, nested classes are an integral part of the design, such as for the example patterns in Paper II, and we have after the publication of Paper II done some additional work on this matter. Below we suggest a set of rules for incorporating nested classes in PT for Java.

Nested classes can be defined for template classes as for normal Java classes. Nested template classes follow the same rules as other template classes, as well as the rules for nested classes in Java, except when explicitly noted in the bullet points below.

- A nested class can only be merged with another nested class at the same level of nesting. This rule prevents non-nested classes from being merged with nested classes, which could lead to several issues, for instance that `new` statements in template code could become invalid. Furthermore, it ensures that no "gaps" are introduced in an existing nesting hierarchy.
- If a nested class is merged with another nested class (at the same nesting level), the enclosing classes, given that they are not the same, must also be merged (recursively, if the nesting level is greater than 1). Without this rule, a class could end up being nested inside several unrelated classes (that are not nested in each other), which obviously makes no sense in context of the Java programming language.
- Analogously to the rule above, an addition class must be at the same level of nesting as the class to which it adds. Thus, nested addition classes are needed in order to add to nested template classes.
- It is allowed to merge outer classes without merging any nested classes the outer classes might have. Nested classes must be renamed if there are other nested classes with the same name that would otherwise lead to unintended merges.
- Static nested classes and non-static nested classes (often simply called inner classes) cannot be merged with each other. Java treats static and non-static nested classes differently, and merging them is thus undesirable.
- Nested classes may only have parameterless constructors (except for the implicitly defined parameter to the outer class for non-static nested classes). This is a restriction that we have introduced in order to simplify the mechanism both from a conceptual and an implementation-centric view. The restriction can be lifted at a later stage if deemed worthwhile.

- Required types (introduced in Papers VI) cannot be nested inside template classes. Required types are intended to be template-level constructs.
- Required types cannot contain nested types. This is a restriction that we have introduced in order to simplify the mechanism both from a conceptual and an implementation-centric view. The restriction can be lifted at a later stage if deemed worthwhile.
- Nested classes cannot contain Aspect-oriented declarations (pointcuts, advice). This rule is chosen in order to make the translation of AOP constructs into other, existing, AOP mechanisms more straight-forward.

These rules are intended to ensure that allowing nested classes in PT will not lead to inconsistent programs, and that nested template classes can be utilized in a "natural" way in PT. That is, nested classes can be used in much the same way as ordinary template classes, with support for refinement, merging, renaming, etc. Note, however, that this does not entail any form of family polymorphism or dependent types; the nested template classes end up as ordinary Java nested classes when a template is instantiated in a package.

As can be seen in the syntax overview in Figure 5.2, the syntax for renaming template classes also supports renaming of nested classes.

Generics and pluggable patterns. In our example implementation of the Memento pattern in Paper II, an abstract class `State` is utilized to signify that there needs to be *some* class that embodies the state of the `Originator` object, without prematurely specifying in any further detail what that state is.

In Paper VII (Chapter 13), we introduce a new variant of generics in PT called *required types*. Required types in templates allow several classes to be parameterized by a common generic type, and furthermore allow such generic types to propagate to instantiating templates if they are not concretized in the instantiation. This allows for refinement and adaption of generic parameters in the same way as for ordinary template classes.

For implementing the Memento design pattern, a required type would probably be a better option than an abstract class, and an implementation (also utilizing the `tabstract` modifier mentioned above) could then look like the following:

```
template MementoPattern {
  required type State { }

  public class Originator {
    public Memento CreateMemento() {
       return new Memento(GetState());
    }
```

```
    public tabstract void SetMemento(Memento m);
    protected tabstract State GetState();
  }

  public class Memento {
    assumed Memento(State state);
    Memento(State state) { this.state = state; }
    private State state;
  }
}
```

Utilizing a required type instead of an abstract class gives the additional benefit of allowing the use of an existing class to store the state (for instance the standard `java.util.HashTable<K,V>`), but it also makes it possible for the programmer to choose that the state should be stored by objects of the (instantiated) `Originator` class (i.e., the state is stored by objects of the same class as the object to which it applies).

The remaining design patterns from [53]. In [53], Gamma et al. describe in total 23 design patterns, while only three of these were treated in Paper II. However, we have also implemented each of the remaining 20 design patterns from [53] with PT. The source code for all 23 pattern implementations in PT can be found at the SWAT project's software page: `http://swat.project.ifi.uio.no/software/`. For each of the pattern implementations available online, there is also a short text file that summarizes the main points with regard to the usage of PT-specific features in each implementation. We will not discuss these implementations in isolation in this section. Instead, we will in the following discuss the PT implementations in context of corresponding implementations in plain Java and AspectJ from [65].

Comparison to the pattern implementations from [65]. In their 2002 paper [65], Hannemann and Kiczales discuss design pattern implementations in Java and AspectJ, and compare their relative strengths with respect to these implementations. They provide source code available for download, allowing us to directly compare an AspectJ implementation with our PT implementation (indeed, for our implementation we used the AspectJ and/or Java implementations from [65] as a starting point, and we tried to keep our implementation as close as possible to the original source, for ease of comparison). The source code for the AspectJ and Java implementations discussed in [65] can be found here: `http://www.cs.ubc.ca/labs/spl/projects/aodps.html`. In the following we will briefly discuss the PT pattern implementations in context of the implementations (both AspectJ and, where relevant, Java versions) done by Hannemann and Kiczales, and try to summarize important differences between them.

The discussion assumes that the reader is, at least to a certain extent, familiar with both AspectJ and the work from [65]. Prior exposure to the work by Gamma et al. would also be beneficial in order to fully understand the following sections.

Added type safety. *Applies to design patterns: Mediator, Command, Memento, Observer.* The PT implementation of these patterns allow for more type safe code compared to their AspectJ counterparts from [65]. This is due to the fact that in the AspectJ variants, roles are typically defined by empty "marker" interfaces, such as e.g. the role `Colleague` in the Mediator pattern, and the code specific to concrete participants must thus use casts from the interface types to the concrete type.

In PT, this approach is often not necessary. Instead of declaring roles as empty interfaces, we can define them as classes that are merged with the actual concrete pattern participants, and the methods typed with abstract roles in templates will thus be typed with concrete participants in the final program.

Reduction of central state. *Applies to design patterns: Command, Iterator, Mediator, Observer, Strategy, Singleton.* Several of the pattern implementations in AspectJ utilize global mapping structures to model relationships between participants in the patterns. For instance in the Mediator pattern, relationships between the roles *Colleague* and *Mediator* are stored in a hash map `mappingColleagueToMediator`, which is accessed through the aspect using the static `aspectOf()` method, e.g.

```
MediatorProtocol.aspectOf().setMediator(colleague, mediator)
```

This basically entails accessing the aspect as a singleton object, and the hash map is thus global state. Global state can make a program hard to reason about, and it might thus be considered unwanted.

For the PT implementation of these patterns, the mappings are instead localized in the relevant participant classes, and these classes are subsequently merged with domain classes, or refined using addition classes. For the Mediator pattern implementation in PT, the statement corresponding to the use of `aspectOf()` shown above is the following:

```
colleague.setMediator(mediator)
```

With the approach above, no mapping is necessary, since the mediator for each colleague is stored directly within the colleague itself. Note that this does not necessarily imply that the pattern code is tangled with the concrete participant code; the pattern code is kept separate in template classes, and merged (weaved) with participants upon instantiation at compile-time.

The Composite and Flyweight pattern implementations in AspectJ also utilize the static `aspectOf()` method, but for these patterns we were unable to get around the need for a globally accessible "bookkeeper" role. We thus ended up with implementing a public class with public static methods for this purpose, mimicking the role of the singleton aspect object in the AspectJ implementations.

Reliance on specific interface implementations. *Applies to design pattern: Iterator.* In the AspectJ implementation, the Iterator pattern includes an interface named `SimpleList` that defines the basic list operations `count()`, `append(Object)`, `remove(Object)`, and `get(int)`. The implementation is dependent on concrete classes implementing this specific interface.

The PT implementation of the Iterator pattern is less tightly coupled to a concrete interface, as it uses a generic specification in the form of a required type (see Paper VII, in Chapter 13, for more on required types). This leads to better reusability for the PT implementation, as the code that utilizes the reusable pattern does not have to conform to a specific (nominal) interface. (Note that ordinary Java generics would not be of much help here, since the implementation would still be confined to a nominal bound.)

Compile-time warnings/errors. *Applies to design patterns: Façade and Builder.* In the AspectJ implementation of the Façade pattern, an aspect declares a compile-time warning if code tries to access the encapsulated object without going through the façade. In PT there are no constructs for declaring compile-time errors. However, in this concrete case, there is no need to do that either, as the PT façade implementation utilizes addition classes to completely encapsulate the classes having the *Subsystem* role in the pattern, and override their methods, thus making it *impossible* to call any methods there directly without going through the façade.

In the Builder pattern, a similar approach for declaring errors is used in the AspectJ implementation to restrict access to an internal `representation` variable, that needs to be declared public in that implementation since it is defined on an interface. In PT, this variable can be declared as private or protected, and such an error declaration is thus not needed (however, access control is not fully specified in PT yet, see Section 12.3.8 for a discussion).

Use of AspectJ's intertype declarations. For a majority (13/23) of the pattern implementations in AspectJ from [65], a set of roles, typically in the form of interfaces, are superimposed on participant classes by utilizing the `declare parents` construct of AspectJ. Implementation of the general pattern roles are done with AspectJ's mechanism for default implementations of interface methods.

In the Java implementation of the patterns, this is typically solved by making the pattern class an abstract base class from which the participant(s) in the pattern need(s) to inherit. This has one important drawback, which is that the pattern classes can then not be part of another inheritance hierarchy. Obviously, this makes programming of certain kinds of applications harder, especially for Java GUI applications where you typically must inherit e.g. from `JFrame` or similar classes.

In PT, the superimposition of roles can be handled by a template class that is merged with the domain participant class. If a role should be superimposed on multiple participants, the template can be instantiated multiple times. Thus, a class can have default implementations for pattern code and still be merged with a class that is part of an inheritance chain. However, this approach has one particular drawback, which does not really manifest itself in the example pattern implementations, but that nonetheless is a real limitation: if a template class represents a role that is superimposed onto more than one participant class, and the template at the same time has other classes that there must only be one of (e.g. because of typing/polymorphism issues), then the approach of instantiating the template multiple times will not work.

In such scenarios, the AspectJ solution of intertype declarations and default interface implementations is more flexible.

The next version of Java (at the time of writing), version 8, is scheduled to include a feature known as *default methods* [34] in interface declarations. This allows the programmer to add default implementations to method signatures defined in an interface declaration. Adding the same capability to PT would to a large degree resolve problems with superimposing roles onto many participant classes, and seems to be a valuable future addition to the mechanism.

Use of pointcut and advice*. Applies to design patterns: Mediator, Observer, Proxy, Singleton.* Only for the Mediator and Observer patterns did we utilize pointcuts and advice (as proposed in Paper I) in the PT implementations, while the AspectJ implementations use these constructs for 11 of the total 23 pattern implementations. For the remaining 12 patterns, the AspectJ implementations utilize AOP constructs mostly for intertype declarations, in order to assign roles to participants, as discussed above.

In the PT implementations, the use of `tabstract` and `tsuper` declarations paired with class merging and additions can in many cases remove the need for pointcuts and advice. However, sometimes we are unable to express exactly the same semantics.

For instance, for the Proxy pattern, the AspectJ implementation uses join point filtering in the advice (using e.g. "`joinPoint.getThis() instanceof SomeClass`") in order to block calls from certain callers. With the limited AOP features in PT, we do not have this possibility (since there is no explicit join point object available), and a similar feature would have to be implemented e.g. through stack introspection or dynamic proxies (see for instance `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html`).

For the Singleton pattern, AspectJ allows the use of pointcuts to control whether specific subclasses of the singleton pattern are themselves singletons or not. In the PT implementation (as in the Java version), the use of a private constructor effectively precludes writing any subclasses.

Precedence declarations*. Applies to design patterns: Decorator, Strategy.* For some patterns, the participants are sequenced in a specific way, for instance allowing one decorator to decorate the output from another. In AspectJ, this kind of precedence can be expressed using explicit `declare precedence` statements. In the corresponding PT implementation, precedence is (more implicitly) expressed by instantiating one template within another. The end result is similar.

In Java, on the other hand, there are no similar mechanisms for declaring precedence, and it is instead done by making explicit dependencies between objects in the code, for instance taking one participant in as an argument to another. This makes it possible to change the ordering at runtime, even though this is not exploited in any of the pattern implementations.

Summing up. The pattern implementations in PT are comparable to those done by Hannemann and Kiczales for AspectJ in terms of their reusability, pluggability, and

modularity. Each approach has some advantages and disadvantages as discussed above. The PT pattern implementations, and the comparison with corresponding Java and AspectJ implementations, provide some added support for the claim that PT is indeed a viable mechanism for creating reusable, pluggable, code, and furthermore, that the addition of a very limited set of AOP constructs to PT may be worthwhile, even though it was only necessary for a few of the pattern implementations.

It is also worth noting that for many of the PT implementations, even though it is not listed explicitly above, there are still benefits to using PT to implement the patterns compared to their plain Java counterpart. This is especially so with regard to reduction of code tangling and scattering, and furthermore with regard to resolving problems that have to do with the somewhat limited possibilities for code reuse offered by traditional single inheritance, as found in Java and many other languages. Several of the example implementations from [53] utilize multiple inheritance as supported by C++ in order to combine pattern code and domain code, and in the PT implementations, the corresponding implementation typically utilizes class merging for a similar end result, while the plain Java version typically requires more code duplication.

6.3 Paper III: *Groovy Package Templates — Supporting Reuse and Runtime Adaption of Class Hierarchies*

Authors. Eyvind W. Axelsen and Stein Krogdahl. Axelsen is the main author, and was responsible for the initial idea, for the design of the PT mechanism as applied to Groovy [116], for writing the paper and for implementing the prototype for Groovy. Krogdahl provided important feedback and helped shape both the idea and the paper.

Publication. Published in Proceedings of the 5th Symposium on Dynamic Languages (DLS), 2009 [6]. Acceptance rate 37%.

6.3.1 Summary

In this paper, we describe the design and implementation of a variant of PT for the dynamic language Groovy, and show through a set of examples the utility of this approach. This version of PT, called GPT for short, is implemented directly in Groovy itself, allowing the programmer to make use of PT simply by referencing a library. The impementation makes heavy use of Groovy's strong capabilities for metaprogramming.

A main motivation behind this paper was to investigate and substantiate our claim that PT is a viable mechanism for different languages, even though most of the examples discussed in other papers revolve around a Java implementation.

GPT supports most of the core constructs from Java PT, including instantiations, addition classes and merging. The main difference, aside from dynamic typing mandated by the language, is that instantiations in GPT are performed at *runtime*. This opens up for many possibilities by letting the runtime state of the program affect the

class hierarchies resulting from instantiations. An instantiation in GPT thus looks like a normal call in the program. Below is an example of instantiating a template named `T`:

```
def INST = PT.instantiate { template T }
```

The classes (or class hierarchies) resulting from the instantiation above are stored in the variable `INST`. Supposing that the template named `T` contains a class `A`, objects of this class can be created by referring to the template instance variable, e.g.:

```
def a = new INST.A()
```

The paper revisits the Observer design pattern [53] example that we have also discussed in papers I and II, but this time a slightly different approach is made possible due to the dynamic typing of Groovy paired with the capabilities for merge in PT. The problem of superimposing roles onto multiple participants, as discussed in Section 6.2.2, can in the dynamic version of PT be solved just by merging a template class with *multiple* other classes with distinct names, which is discussed in a little more detail below.

6.3.2 Discussion

A variant of PT for a dynamic language, such as Groovy, effectively nullifies some of the claimed benefits of the mechanism for statically typed languages, such as e.g. type safe class merging and renaming (due to a lack of knowledge about bindings at instantiation time), statically checked templates, and compile-time specialization. On the other hand, the freedom of expression typically associated with dynamic languages opens up possibilities that can be utilized also in PT.

Thus, one way to view the dynamic variant of PT up against the statically typed variant is to consider the respective advantages and trade-offs of dynamic versus statically typed languages in general. Typically, in a dynamic language, you trade in relative compile-time safety for a greater degree of expressiveness; with a static type system, some programs that *are* correct (i.e., they would produce the desired results at runtime, without errors) will be ruled out by the type checker [21], because the type checker cannot *prove* them to be type-correct. So is it also with PT: the dynamic version provides a great deal of freedom of expression (and even more so with the extensions introduced in the next paper, Paper IV), but on the other hand, it comes with weaker semantical guarantees.

Merging a class with multiple other distinctly named classes in one instantiation. An important benefit of PT in a dynamic setting, which is also utilized in Paper III, is that one can merge a template class with *a number of* other classes, with distinct names, *in the same instantiation*. That means that statements like "`inst T with C => D, C => E`", which quickly would lead to untypable programs in the statically typed variant, is allowable (under certain conditions, see below) in GPT.

This allows for creating trait-like classes in templates, that can provide functionality that can be common for a number of classes without enforcing a corresponding inheritance hierarchy. For such instantiations, template classes that are merged with more than one other class with distinct names in an instantiation must be abstract (otherwise, what would e.g. `new C()` mean?), without any public static (class) members.

Furthermore, there is an issue if such classes have subclasses. Whether a template class has subclasses (within the template) or not can easily be determined. If a class `C` does have one ore more subclasses, there are basically two approaches that can be taken if `C` is merged with more than one other class. The first option is to disallow such merges, and this is what the GPT implementation does today. Another option would be do designate the first of the target classes in the instantiation as the superclass of the subclasses in question after an instantiation. Thus, given the statement `inst T with C => D, C => E`, any subclasses of `C` would after instantiation have `D` as their superclass. Since both `D` and `E` will have all the contents of `C`, this is safe. This is arguably the more flexible solution, but it might be confusing, especially since the order of instantiations in PT in general is insignificant.

Runtime instantiations. The ability to perform instantiations at runtime gives the developer a great degree of freedom to adapt the *structure* of the program according to conditions that can only be known when the program is actually running, presumably at a client's site. This allows for writing programs that treat entire APIs in a polymorphic fashion, and it allows runtime conditions to be used as decision-points for interacting with different instances of such APIs. Furthermore, it can be used to implemented other wide-spread techniques, such as dependency injection and so called "monkey-patching" (runtime class modifications). However, in contrast to traditional monkey-patching, PT provides the important benefit of explicit control of the scope of runtime changes to classes upon instantiation (thus not affecting global state, as opposed to most other dynamic languages' support for such modifications).

Even though it is not shown explicitly in Paper III, having a runtime instantiation specification entails, obviously, that the template name and other parts of the instantiation may be supplied as variables instead of being written as explicit constants in the call to the `instantiate` method.

Paper IV builds on the concepts for runtime instantiations introduced in Paper III to provide an even more flexible approach to adapting the program's structure.

Dependency on Groovy specifics. The constructs in the paper are presented in terms of how they are applied to and implemented in the Groovy language. Applicability to other dynamic languages will of course depend on the semantics of the language in question. The implementation makes use of Groovy's reflective capabilities, and similar capabilities would be needed in order to make a comparable implementation in another dynamic language.

Furthermore, the implementation relies on the fact that subtype relationships between classes and interfaces is static in nature in Groovy, i.e. a class cannot at runtime change its superclass or its interface implementations. For languages that allow

e.g. using runtime expressions to evaluate the superclass of a given class, such as in Newspeak [28], there are several open issues. For instance, how would one prevent multiple inheritance in the face of merge, when one does not know which superclass will eventually be bound to the merged classes? Furthermore, if, like in Newspeak, there is no global namespace, how would one get hold of templates in the first place?

For Newspeak and languages with similar capabilities, it seems that even more checking would have to be deferred to runtime, including multiple inheritance checking from merging. Still, these, and several other issues, remain open for potential future research.

Prototype implementation as an internal DSL. The prototype implementation that accompanies this paper is built as a Groovy class library that provides a small internal DSL for instantiating templates. That the DSL is internal means that it is implemented within the host language, here Groovy, as opposed to having a specialized parser and compiler/code generator for the DSL. In other words, an internal DSL is provided as an API in the host language. The feasibility of such an approach depends much on the malleability of the syntax of the host language, and its support for meta-level programming. The Groovy language is relatively well equipped for such a task, providing method calls without parenthesis (which we utilize for providing a cleaner syntax), closures with alternate resolution strategies (which we utilize for executing a closure in a different environment than the one it was created in, aiding in creating a flexible, command-like syntax), and meta-level control of *message not understood*-scenarios (which we utilize for creating pseudo-keywords).

The case for utilizing an internal DSL for such implementations is strengthened by the fact that the user does not need any additional compilers or preprocessors in order to use the mechanism. Furthermore, since the DSL itself is Groovy code, the barrier for adoption would seem to be lower.

However, the current Groovy implementation also leaves a few things to be desired. To begin with, the addition classes for this prototype are specified at the same level as the class in which the instantiation statement (call to `PT.instantiate`) is declared, while it would perhaps be more natural to have the addition classes as part of the instantiation statement itself, since their utilization depends on the dynamic behavior of the program. The reason for not doing this is that Groovy at that time did not support nested classes (support has since been added to the language), and the choice was then between keeping the addition classes at the same level as ordinary classes, or using another construct, e.g. a dictionary with methods, for implementing additions. This is of course one of the trade-offs we had to make for utilizing an internal DSL: you need to play by the rules of the host language.

On the other hand, using plain Groovy for instantiations gives us the full power of the Groovy language to manipulate instantiations at runtime, and this is utilized heavily in Paper IV, which builds directly upon this paper.

Combination of static and dynamic typing. The Groovy language supports, to a certain extent, declarations that are statically typed. However, since the language is

mainly a dynamically typed one, the Groovy compiler cannot reliably detect e.g. invalid calls to statically typed methods at compile time, or perform major performance optimizations based on static typing (since new attributes may be added to classes and objects at runtime) [31, 63]. Groovy 2.0 (which was made available after Papers III and IV were written) allows the programmer to annotate Groovy code with `@CompileStatic` [78], but this (obviously) changes the runtime semantics of Groovy to be closer to that of Java.

In papers III and IV, and in the PT implementation for Groovy, we have not considered the statically typed parts of Groovy. If GPT were to support the optional static typing of Groovy, we would have had to impose certain restrictions, perhaps most notably we would have to forbid the merging of a template class with multiple other classes. There would also have to be severe restrictions on the allowable instantiation strategies discussed in Paper IV, if we were to ensure type safety for statically typed declarations.

6.4 Paper IV: *Controlling Dynamic Module Composition Through an Open and Extensible Meta-Level API*

Authors. Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller Pedersen. Axelsen is the main author, and was responsible for the main idea, for the design and implementation of the constructs described, and for writing all the text of the paper. Krogdahl and Møller-Pedersen provided important feedback and helped shape both the idea and the presentation in the paper.

Publication. Published in Proceedings of the 5th Symposium on Dynamic Languages (DLS), 2010, as part of the SPLASH 2010 collection [10]. Acceptance rate 29%.

6.4.1 Summary

This paper builds upon and extends the mechanisms and concepts that we developed for package templates in Groovy (GPT) in Paper III. Paper IV takes the idea of packages (or *modules*) that are adaptable at runtime one step further, and introduces a meta-level framework for describing such adaptions. We show that an extended version of GPT can support varying composition semantics through the use of meta-level *strategies* that exploit an API that gives access to instantiation specifications and provides hooks at specified points, called *interception points*, throughout the instantiation process.

The GPT mechanism is different from most mainstream approaches to runtime class adaption in that it provides a well defined scope at runtime (i.e. each instantiation) to which changes to and compositions of template classes are confined. The same applies to changes to the composition semantics, in that they can be given an explicit local scope, and will thus not interfere with other parts of a program (even other instantiations of the same templates). This further entails that a program can contain a variety of different composition semantics at the same time, and these can even be changed

throughout the lifetime of a program, based on runtime conditions. For instance, the semantics of conflict resolution when classes are merged can be changed by the programmer, and can depend on runtime conditions.

In addition to supporting variable composition semantics, the mechanism also supports introduction of new composition primitives by the programmer. In the paper, this is exemplified by defining a small aspect-oriented extension to the instantiation statement.

Finally, we introduce the notion of templates that can be aware of, and take active part in, their own instantiation process. A template can define meta-level classes that are applied as strategies by the instantiation. This is exemplified by an implementation of the Active Object design pattern [80].

6.4.2 Discussion

One could argue that the approach that we describe and discuss in Paper IV is merely applicable as an extension of the (arguably somewhat esoteric) GPT mechanism, and not as such something that is applicable to a broader range of scenarios. However, we believe that the converse is true. First, the argument would be that runtime modifications to classes is indeed useful, and this is supported by the fact that a large number of popular languages like Ruby, Python, JavaScript and, of course, Groovy have mechanisms for this. Furthermore, such modifications are heavily exploited by popular mainstream frameworks such as e.g. Rails [104] and Grails [115].

The stance that control over the scope of such modifications is important is, at least anecdotally, supported by the large number of patches to frameworks such as the aforementioned two that deal with issues due to so-called "monkey-patching", i.e. the runtime modification of existing classes or objects, and the problem often seems to stem from the fact that there are conflicting modifications done by different libraries that are both used in the same application[2]. Utilizing GPT (including the extensions from Paper IV), it is in our opinion at least somewhat easier to control the ramifications of a change to a class, due to a well-defined instantiation scope, so that each subset of a solution can have its own version of a particular library or functionality, without unknowingly affecting other parts.

The claim that there is a real need for mechanisms that allow the developer to compose elements of code to form new elements, typically in an unanticipated manner, is, at least partially, supported by the large amount of research in this field.

So, that leaves the issue of semantic control over the composition process. If we take for granted that runtime class modifications (and the composition of such) are indeed desirable, the questions are: is there a need for runtime control over the composition semantics, and is the presented approach of a meta-level API a general approach to this problem?

Different mechanisms for modularity provide varying composition semantics, for instance with regard to precedence, interactions and composition rules. Just the fact

[2]See e.g. `https://github.com/rails/rails/commits/master` for a look at the latest commits to the Ruby on Rails codebase.

that many such mechanisms are in widespread use today provide support for the claim that any given semantics cannot fit all problems equally well, and this at least leaves the door open for approaches that aim to provide configurable semantics for composition.

Providing this ability at runtime opens up for modules that can *adapt* their composition semantics to the nature of the environment in which they are deployed, and to other modules with which they are composed.

The generality of our approach is based on the assumption that there are at least a few commonalities shared by many runtime composition mechanisms. First, there is a specification of what is to be composed. For GPT this corresponds to the instantiation specification, while for instance for Groovy's own runtime mixin mechanism, the specification is the classes mentioned as arguments to the `mixin` method. For traits this specification would be the set of traits and potential renames and exclusions specified in the trait list of a class.

Secondly, there is the idea that this specification should be made available through a set of meta-level interfaces that make sense for the mechanism at hand. For instance for traits, the interfaces could expose methods to be called for trait composition, for method exclusions and for conflicts, in much the same way that our framework does. The pre and post instantiation (trait composition) interception points could still apply.

Whether the extension of GPT from Paper IV is applicable to, and indeed helpful for, a broad range of problems can really only be answered by applying the mechanism to such a range, and unfortunately we have not found the time to do this during the work with this thesis. Thus, the overall applicability of the approach, though supported by the examples in the paper, remains a topic for future work.

Performance. When templates are instantiated at runtime, the performance of this operation becomes an important issue for the overall applicability of the mechanism. As we concluded in Paper IV, the performance of the prototype is not particularly impressive, and this would probably rather quickly become an issue in a real-world scenario. However, the prototype was not in any way tuned for performance, and there are some things that with relative ease could be done to improve performance.

The implementation in the paper is based on Groovy version 1.7.1, but later versions have since added several new features and improvements targeting performance. First of all, an upgrade to Groovy version 2.0 would probably in itself provide a performance gain, since the language implementation now takes advantage of the new JDK 7 `invokedymanic` instruction, making dynamic method calls faster [3].

Furthermore, Groovy now supports partial static compilation, which means that certain parts of an application (or certain parts of a class, for that matter), can be statically type checked and compiled. On the JVM platform, this typically results in a significant performance gain. The implementation of PT for Groovy does make heavy use of the dynamic features of the language, but not for all of its implementation. Quite considerable amounts of code could be rewritten to take advantage of statically typed

[3] See e.g. `http://docs.codehaus.org/display/GROOVY/2012/06/28/Groovy+2.0+released`

compilation, leveraging the dynamic expressiveness and paying the corresponding runtime cost in terms of performance overhead only where needed.

6.5 Paper V: *Challenges in the Design of the Package Template Mechanism*

Authors. Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl and Birger Møller-Pedersen. Axelsen and Sørensen are the two main authors of this paper, and wrote most of the text. Krogdahl also made significant contributions to both the conceptual and practical work involved in writing the paper. Møller-Pedersen provided important feedback.

Publication. Published in Transactions on Aspect-Oriented Software Development volume 9, 2012, special issue on modularity constructs in programming languages [12].

6.5.1 Summary

In this paper, we describe and, for several of the more involved issues, discuss, the fundamentals of the PT mechanism, focussing on the statically typed variant of the mechanism. For several of the issues in the design of the PT mechanism there are areas and problems where we believe that we have not yet found the optimal solution, and the paper aims to present the research questions and potential alternative solutions.

Paper V thus represents a continuation and further elaboration and discussion of the initial ideas presented by Krogdahl et al. in [76, 77], as well as a definition of several new concepts that were not discussed in these publications.

The new concepts introduced in this paper are:

- Code in an addition class can refer to method declarations in template classes to which the addition class adds, by using the `tsuper` ("template super") keyword. Thus, the original implementation for a template class method overridden in an addition class can be reached, in a way similar to ordinary `super` calls in Java. Correspondingly, constructors in template classes can be called through `tsuper(...)` calls. This is discussed in Section 12.3.2.

- The `tabstract` ("template abstract") modifier is discussed in Section 12.3.4. It is a parallel to the `abstract` modifier for methods, but while `abstract` designates an abstract definition that *may* be concretized by a subclass (or an addition class), the `tabstract` modifier designates an abstract definition that *must* be concretized in a corresponding addition class (in the transitive sense), at the latest when the template is instantiated in a package. The `tabstract` modifier cannot be applied to class declarations, only to method signatures (and, if the AOP extension described in Paper I is utilized, also to pointcut and advice signatures). This entails that it is safe to create new objects of classes with `tabstract` declarations, since they will be concrete at the latest in the resulting package.

- A construct that allows templates to be used as parameters to other templates in an instantiation is introduced in Section 12.3.7 of this paper. This feature basically allows a template to abstract over variations of other templates specified as parameters with *template bounds*. This feature allows for quite a flexible solution to many problems related to independent extension and composition, such as the Expression problem [134].

In addition to introducing new concepts to PT, the paper also discusses several existing ones (from previous papers) in more detail, such as virtual methods and algorithms for their lookup through addition classes and ordinary super classes, constructors, name conflicts because of merging, access modifiers, subclasses to classes external to a template, and merging of classes with superclasses.

6.5.2 Discussion

In this section, we will discuss some of the concepts from the paper in a little more detail. A significant part of the discussion will be centered around template parameters to templates, and their semantics and interaction with other features of PT, as well as their limitations.

We will also briefly discuss some issues with the `tsuper` construct, and some issues with constructors in relation to template classes and addition classes.

Templates with Java-style type parameters. The type parameters discussed in Section 12.3.7 of Paper V are subsumed by the approach presented in Paper VI, which we call *required types*. We will therefore not discuss the type parameters from 12.3.7 any further here. For more details on required types, see Paper VI in Chapter 13, as well as the discussion in Section 6.6.

Semantics of template parameters to templates. Paper V does not go into much detail when it comes to template parameters to templates, so a brief discussion on the semantics of that construct seems appropriate here. At the core of the approach in the following is that the semantics of templates with template parameters can be defined through a transformation to templates without template parameters. Semantics as for ordinary templates (i.e., without template parameters) then applies; this semantics is treated through a translation to plain Java in Paper VII.

An important principle in the following is that each template supplied as an actual template parameter in an instantiation will be instantiated exactly once (without the programmer having to explicitly state that). There are, however, alternative approaches that are also both consistent and potentially useful. For instance, a seemingly natural approach could be to allow each parameter to be instantiated zero or more times by the programmer (by using ordinary `inst` statements within the bodies of parameterized templates). This provides extra flexibility, but also a certain degree of added complexity. We will get back to this a little later, but for now, the rule is as mentioned above, that each actual template parameter will be instantiated exactly once.

We recall the syntax for template definitions (with or without formal parameters) and instantiations (with or without actual parameters), which is defined in Chapter 5. Before we go into details on the semantics, we consider a simple example to illustrate how templates can be used as parameters to other templates. First, we define a parameterized template `Tp`:

```
template Tp<template X inst Pp> { ... }
```

In this example, the name `X` is a formal parameter, and its *bound* is a template named `Pp`. This means that when `Tp` is instantiated, `Pp` or a template with an *explicit instantiation* (see below) of `Pp` (transitively) can be supplied as an actual parameter. Thus, given e.g. a template of the form "`template Px inst Pp { ... }`" (where `Pp` is explicitly instantiated), we can instantiate `Tp` as follows:

```
inst Tp<Px> with ... ;
```

Explicit instantiations. As mentioned in Chapter 5, an *explicit* instantiation is an instantiation that occurs in the header of a template declaration, as opposed to within the template body. Note that there might be more than one such instantiation. A special restriction applies to explicit instantiations, which is that no renames are allowed. The reason for this is that a template that specifies another template as bound for one of its formal template parameters, should be able to rely on the fact that all names declared in the bound are visible in the actual parameter as well.

As long as this restriction is upheld, a template with explicit instantiations has the same semantics as a template with the corresponding instantiations written inside the template. That is,

`template TE inst` U_1`, ...,` U_N `{` *<template-body>* `}`

is equivalent, in terms of the classes "produced" by the template, to a corresponding template `TE'`:

`template TE' { inst` U_1`; ...; inst` U_N`;` *<template-body>* `}`

Note that U_1, ..., U_N may themselves be parameterized (and if so, should be listed in the instantiation with actual parameters). Furthermore, since renaming of classes from an explicit instantiation is not allowed, the classes in U_1, ..., U_N *must* have unique names (they will not be merged).

Instantiations with actual template parameters. An instantiation that supplies one or more actual parameters to a template `TA` (which must declare corresponding formal parameters) is semantically equivalent to an instantiation of a template `TA'` with equal contents as `TA` and no formal parameters declared, where the templates that are supplied as actual parameters to `TA` are instantiated directly within `TA'`. That is, an instantiation of the following form

```
inst TA<P1 with W1, ..., PN with WN> with W;
```

where `P1, ..., PN` are template names (that may themselves be parameterized), and `W, W1, ..., WN` are `with`-clauses, is semantically equivalent to

```
inst TA' with W;
```

given that the template `TA'` is of the form

```
template TA' { inst P1 with W1; ...; inst PN with WN;
   <template-body> }
```

and that the *template-body* of `TA'` is equal to that of `TA`. Note that the `with`-clauses `W1, ..., WN` cannot change any names defined in the bounds of the formal parameters for which `P1, ..., PN` are supplied as actual parameters (since code within the parameterized template might depend on these names). Such changes can, however, be made in the outer `with`-clause `W` (and corresponding changes will then be made to any use-sites within in the parameterized template).

As indicated above, if the actual parameters supplied in the instantiation of `TA` are themselves parameterized, the transformation is repeated for the instantiations in `TA'`, i.e. for the instantiations of P_1, ..., P_N.

If an actual parameter P_k to `TA` is used as an actual parameter to an instantiation within the body of `TA`, an instantiation of P_k will not be inserted in the body of `TA'`, since the template instantiation that the actual parameter is passed on to will in turn perform this instantiation (or pass the parameter on to yet another template instantiation).

If a template has both a template parameter and an explicit instantiation of that parameter, like in the Expressions example from Section 12.3.7 of Paper V, there will only be one instantiation of that parameter.

To illustrate how the translation to ordinary PT code (without template parameters) can be done, we consider the familiar Expressions example again. Figure 6.2 contains the original parameterized PT code, and Figure 6.3 contains corresponding code transformed to ordinary templates.

As we can see from the translated version in Figure 6.3, the resulting templates are straightforward PT code, which can subsequently be transformed to plain Java code as described in Paper VII.

Static correctness of templates with formal template parameters. A template with one or more formal template parameters can be semantically checked for correctness with the following approach: Create a corresponding template with the same body, but without any template parameters. For each formal parameter in the parameterized template, insert an instantiation of the template used as the parameter's bound in the parameterless template (unless the parameter is used in another instantiation or bound, as outlined above). Then, the parameterless template can be checked by utilizing simple

```
template Expressions {
   abstract class Exp { }
   class Plus extends Exp { Exp left, right; }
   class Num extends Exp { int value; }
}

template PrintExpressions<template E inst Expressions> inst E {
   class Exp adds { abstract void print(); }
   class Plus adds {
      void print() { out("("); left.print(); out("+");
                     right.print(); out(")")}
   }
   class Num adds { // extends Exp
      void print(){ out(value);}
} }

template ValueExpressions<template E inst Expressions> inst E {
   class Exp adds { abstract int value();}
   class Plus adds {
      int value() { return left.value() + right.value(); }
   }
   class Num adds {
      int value() { return value; }
} }

template MultExpressions<template E inst Expressions> inst E {
   class Mult extends Exp { }
}

package CombinedExpressions {
   inst MultExpressions<ValueExpressions<
      PrintExpressions<Expressions>>>;

   class Mult adds {
      void print() { out("("); left.print(); out("*");
                     right.print(); out(")")}
      int value() { return left.value() * right.value(); }
} }
```

Figure 6.2: The Expression example from Section 12.3.7 of Paper V.

```
template Expressions {
   abstract class Exp { }
   class Plus extends Exp { Exp left, right; }
   class Num extends Exp { int value; }
}

template PrintExpressions' {
   inst Expressions;
   class Exp adds { abstract void print(); }
   class Plus adds {
      void print() { out("("); left.print(); out("+");
                     right.print(); out(")")}
   }
   class Num adds { // extends Exp
      void print(){ out(value);}
} }

template ValueExpressions' {
   inst PrintExpressions';
   class Exp adds { abstract int value();}
   class Plus adds {
      int value() { return left.value() + right.value(); }
   }
   class Num adds {
      int value() { return value; }
} }

template MultExpressions' {
   inst ValueExpressions';
   class Mult extends Exp { }
}

package CombinedExpressions' {
   inst MultExpressions';

   class Mult adds {
      void print() { out("("); left.print(); out("*");
                     right.print(); out(")")}
      int value() { return left.value() * right.value(); }
} }
```

Figure 6.3: A transformation of the example from Figure 6.2 into ordinary templates without template parameters or explicit instantiations.

syntactical transformations and standard Java rules as described in Paper VII. If this template is correct, then the original template is correct as well.

Template parameters to templates and name clashes. When instantiating a parameterized template, the programmer may supply actual parameters that contains names that were not known in the bound of the formal parameter. Because of this, name clashes may easily occur; for instance, consider the following simple example:

```
template TBase {
   class A { }
   class C { }
}

template V<template X inst TBase> {
   class A adds {
      void m() { ... }
   }

   class B { ... }
}
```

Template `TBase` is a template with two classes, `A` and `C`, which in this example are both completely empty. The template `V` is parameterized by `X`, which has bound `TBase`. In `V`, a new class `B` is defined, and a method `void m()` is added to `A`.

Below, we consider a template `U`, that has an explicit instantiation of `TBase`, which entails that it might be used as an actual parameter where `TBase` is the bound for the formal parameter. The template `U` also (coincidentally) adds a class `B`, and a new method `void m()` to `A`. Note that the templates `U` and `V` are completely independent, and each of them might have been written without any knowledge of the other.

```
template U inst TBase {
   class A adds {
      void m() { ... }
   }

   class B { ... }
}
```

If we were to instantiate `V` and supply `U` as the actual parameter, i.e. "`inst V<U>`", there would be name clashes both for the two definitions of a class named `B`, and similarly for the method signature `void m()` in `A`. Both of these must be handled, as for ordinary parameterless instantiations, through renaming specified by the programmer in the instantiation of `V`.

Note that if the class `A` in `TBase` had declared a method `void m()`, the instantiation "`inst V<U>`" would result in `U.A.m` being an override of `TBase.A.m`, and

`V.A.m` being an override of `U.A.m`. No renames of `m` would thus be mandatory. This follows from the semantics presented above, that describes how templates with explicit instantiations and parameterized templates can be translated to ordinary templates (without template parameters or explicit instantiations) through relatively simple transformations.

For a parameterized instantiation, renames can be specified after the actual parameter in the `inst` statement, as shown in the syntax overview in Figure 5.2. However, there are some rules that must be followed when specifying such renames:

- Names (of classes, methods, etc) that are declared in the template used as bound for the formal parameter (such as names stemming from template `TBase` in the example template `V` defined above) may *not* be renamed inside the actual parameter specification. For instance, consider the following instantiation:

  ```
  inst V<U with A=>A(m()->n), B=>Y /* illegal:  C=>Z */ >;
  ```

  The rename of the method `m` to `n` is legal because `m` is first defined in the template `U` (in the addition class `A`). Likewise, the rename of the class `B` to `Y` is legal for the same reason. A rename of class `C`, however, is illegal within the parameter specification, because `C` is first defined in the bound for the parameter, which is `TBase`. Outside of the parameter specification, on the other hand, `C` can be renamed, as mentioned above. In other words, the following is legal:

  ```
  inst V<U with A=>A(m()->n), B=>Y> with C=>Z;
  ```

- Names defined in a template used as an actual parameter may not clash with names declared within the parameterized template.
- A template that declares a formal template parameter may not rename any of the names declared in the bound for this parameter.
- Likewise, a template with an explicit instantiation clause (such as `U` has of `TBase` above) may not, as mentioned above, rename any names declared in the explicitly instantiated templates.

These rules entail that two templates containing conflicting names cannot both be used as explicit instantiations for one template declaration, neither can they both be used as bounds for formal template parameters in one template declaration.

Issues with allowing the programmer to more freely instantiate template parameters. As mentioned in the introduction to the discussion about the semantics of instantiations of parameterized templates, an alternative approach to the one described above (where each template parameter is instantiated exactly once by the PT compiler), is to allow the programmer to use ordinary `inst` statements within a template body

to instantiate template parameters. This provides added flexibility, but also introduces some complications. We will consider a couple of these below.

In the example program below, we assume that there are no `inst` statements inserted by the PT compiler, but that the programmer is free to insert his/her own. Furthermore, we assume that the programmer of `TBase` and `TFree`, at the time of writing these, does not know of the template `TSub` and its subsequent usage in `P`:

```
template TBase {
  class A { }
  class C { }
}

// note that we instantiate TX twice below
template TFree<template TX inst TBase> inst TX {
  inst TX with A => AA, C => CC;
}

template TSub inst TBase {
  class B { }
}

package P {
  inst TFree<TSub with B => BB>;
}
```

The problem that this example illustrates is that templates like `TFree`, which would appear reasonable at the time of writing, actually preclude almost any extension of `TBase` to be usable as a parameter (and thus render the entire parameterization nearly useless). The reason for this is that there will be a conflict with two `B` classes, one stemming from the explicit instantiation of `TX` in the header of `TFree`, and one from the instantiation of `TX` within the body of `TFree`. We could perhaps have resolved this e.g. by introducing more elaborate mechanisms for renaming, but this comes with its own set of complications and potential pitfalls.

Another, somewhat related, issue arises when we have a more complicated set of template dependencies, where one template parameter depends on another. Again, we assume that there are no `inst` statements inserted by the PT compiler:

```
template G { class A { ... } }

template H1<template X inst G> {
    // Note that there is no inst of X here.
    // H1 is thus empty.
}
```

```
template H2<template X inst G> inst H1<X> {
   inst X; // H2 will have (at least) the class A from G
}

template I<template X1 inst G, template X2 inst H1<X1>> {
   inst X1;
   inst X2; // will there be an inst of G in here?
}
```

In this example, there are a couple of things worth noting. First, we cannot know within `I` whether the class `A` from `G` will be duplicated or not, since that depends on which template is supplied as an actual parameter for `X2` (i.e., `H1<G>` or `H2<G>`). Second, we cannot alleviate this problem by not instantiating `X1`, since then we cannot use the class `A` at all.

Both of the issues above are related to having explicit instantiations of formal parameters. We could have disallowed such instantiations altogether, but this seems to rule out several useful idioms of template parameter programming, as exemplified by the Expression example discussed above and in Paper V. Another potential solution to this problem could be to create special restrictions for template parameters used in explicit instantiations, however, that would further complicate the mechanism.

As the interdependencies between parameterized templates get more complex, the toll on the programmer to keep track of them increases. Even though the semantics presented in the beginning of this section, where the PT compiler keeps track of this and every parameter is instantiated exactly once, is in many ways somewhat limited, it still seems like a better approach, at least at the current stage in our research in this area.

Issues with `tsuper` calls. The `tsuper` construct, introduced in Paper V (Section 12.3.2), provides the programmer with added flexibility to call overridden method implementations. Correspondingly to how methods from superclasses can be called directly with `super`, methods from classes to which the current class is an addition can be called directly with `tsuper`. However, mixing `tsuper` and `super` calls is not without pitfalls. For instance, consider the following template:

```
template T {
   class A {
      void m() { ... }
   }
   class B extends A {
      void m() { ... super.m(); ... }
   }
}
```

In isolation, this looks like a completely reasonable template. However, what if we now introduce the following template, that makes use of `T`:

```
template U {
   inst T;
   class A adds {
      void m() { ... tsuper.m(); ... }
   }
   class B adds {
      void m() { ... tsuper.m(); super.m(); ... }
   }
}
```

This template also looks reasonable, when considered in isolation. The method `m` in the addition class `B` calls `tsuper.m` in order to include the implementation from the template, and `super.m` in order to include the implementation from its superclass. However, according to the algorithms for `super` and `tsuper` lookup presented in Section 12.3.2 of Paper V, this will actually lead to the methods `m` in both `T.A` and `U.A` being called twice. The reason for this is as follows: When `tsuper.m()` is executed in `U.B.m`, this will call `T.B.m`. The `super.m()` call there will resolve to `U.A.m`, and the `tsuper.m()` there will call `T.A.m`. The next statement in `U.B.m` is then `super.m()`, and the execution of this statement will resolve to `U.A.m`, and the `tsuper.m()` call there will finally lead to another call of `T.A.m`. This resulting sequence of calls is probably not what the developer of either template intended.

This problem is very similar to that of mixing `super` and `original` calls in ClassBox/J [18], however, the ClassBox/J paper is not entirely clear (as far as we could understand) as to whether `super` calls in class refinements are actually allowed or not, and the implementation of ClassBox/J (as of September 2012) appears to crash when such code is compiled.

For PT, disallowing `super` calls in methods that may be overridden in addition classes is indeed an option that would prevent this problem, but this would also quite severely restrict the expressiveness of the mechanism. Another option would be to have a compile-time warning based on call graph analysis, however, such warnings cannot take runtime conditions into account, and can as such never provide any guarantees.

For the current version of PT, we have thus opted to leave it to the programmer to be aware of and avoid this particular problem.

Template parameters and tsuper. If an addition class within a parameterized template adds to a template class defined in the bound of one of this template's formal template parameters, `tsuper` calls within that addition class cannot be statically bound to their target when the template is written. This is because the template used as an actual parameter in an `inst` statement of the parameterized template may define overrides of methods defined in the bound for the formal parameter. Thus, in order to bind these calls the compiler needs to look at the actual `inst` statements, and not only the template definitions in isolation.

Constructors. The use of constructors in template classes and addition classes would exhibit the same problems with multiple calls as discussed above, were it not for the rules introduced in Section 12.3.5. However, as opposed to for methods, it is clear that we want a constructor to be called at most once, and furthermore, that we want at least one constructor to be called for each template class and each addition class. Thus, we can create rules that enforce that `super` calls to constructors are only made in the "outermost" addition classes (necessarily residing in a package), and `tsuper` calls to constructors are made to each template class constructor. In the paper, we refer to this set of rules for constructors as the "backwards E strategy", based on the direction of the constructor calls if one envisions a figure with set of addition classes following to the right of template classes. This strategy provides a way to call constructors for each template class and addition class, and furthermore allows parameter passing between them. The "backwards E strategy" is the currently implemented and supported constructor mechanism.

However, this scheme is, as discussed in the paper, not without its drawbacks. Most prominent of these is perhaps that template classes cannot call their `super(...)`, but must rely on constructors in addition classes in the package to do that. An alternative variant, called the "lying E strategy", is briefly discussed, but this has its own set of issues.

In the paper, we briefly discussed the use of `new` within template classes, and concluded that it would have to suffice to use this only for parameterless constructors. However, we have since, through experimentation with reimplementing Java libraries in PT, found this to be too restrictive. Therefore, the implemented version of the mechanism now supports the declaration of *assumed* constructors. An assumed constructor is just a constructor signature (i.e. it does not have a body), declared in a template class, with the modifier `assumed`. For each assumed constructor, the corresponding package addition class must provide a constructor with that signature. This makes it safe to use the `new` statement in templates to make objects of template classes that have constructors with (or without) parameters.

A recent and interesting approach to object initialization can be found in the Magda language [22]. Magda has a *modular initialization protocol*, meaning that instead of ordinary constructors, mixins (Magda does not have classes) have initialization methods with a list of input *and* output parameters. The output parameters explicitly specify assignment of values to the initialization methods of super mixins. The initialization methods are uniquely identified by also utilizing the name of the mixin itself as a qualifier. A mixin needs not initialize all the parameters of the initialization method(s) of its super mixin(s), the uninitialized values will be propagated to object creation time, if not initialized by another mixin in the chain.

The ideas of Magda are not directly transferable to a system such as PT, mainly because we allow `new` statements in template classes, which entails that a package class must support these constructors explicitly, and because there are two independent dimensions of extension.

However, it would be an interesting topic for future work to consider how an approach based on the initialization methods of Magda could be adapted to PT, and if this could resolve some of the outstanding issues with constructors.

6.6 Paper VI: *Adaptable Generic Programming with Package Templates and Required Types*

Authors. Eyvind W. Axelsen and Stein Krogdahl. Axelsen is the main author, and was responsible for the initial idea, the design of the constructs described, and the programming of the case study described in the paper. Krogdahl provided important feedback and helped shape both the idea and the paper. The implementation of the concepts in the prototype PT compiler were performed mainly by Steinar Kaldager, with guidance and assistance from Axelsen, Krogdahl and Daniel Rødskog.

Publication. Published in Proceedings of the 11th annual international conference on Aspect-oriented Software Development, 2012 (AOSD'12) [8]. Acceptance rate 25%.

6.6.1 Summary

The paper presents an addition to PT called *required types*, and deals with programming and adaption of generic libraries. We refer to PT with required types as PT*r* for short. The essence of PT*r* is that it allows the specification of requirements for template-wide types that are to be supplied at instantiation. This resembles traditional generic parameters as found in Java, or indeed type parameters to templates as briefly described for PT in Paper V (required types are intended to replace the latter). An important distinction between required types in PT*r* and "traditional" generics (as found in Java as well as previous versions of PT) is that the required types *propagate* through template instantiations if they are not explicitly concretized. Furthermore, required types can be adapted in a manner similar to regular PT classes, in the sense that the may be renamed, merged and refined through addition clauses when the template is instantiated. This enables PT*r* to support associated types, constraint propagation, multi-type concepts, adaptable generic constraints, and adaptable generic concepts.

With a required type, the programmer can specify both nominal and structural constraints for the same type. A required type declaration is declared within a template at the same level as ordinary classes, e.g. as follows:

```
template T {
   class A {
      void m(R r) { ... r.run(); ... r.stop(); ... }
   }
   required type R implements Runnable { void stop(); }
   required type U extends java.util.Stack<A> { }
}
```

The type `R` in the example above has both a nominal constraint to implement the `Runnable` interface (from the `java.lang` package), and a structural constraint to implement the `void stop()` method (which is not part of the `Runnable` interface). Ordinary Java generics and required types can be mixed in a template; in the example above, the required type `U` has a nominal constraint that includes the template class `A` as a generic parameter to its bound.

Upon instantiation of `T`, the required types may be concretized by utilizing the `<=` arrow:

```
template T2 {
   inst T with R <= C;

   class C implements Runnable {
      void run() { ... }
      void stop() { ... }
   }
}
```

Note how only the type `R` is concretized in the instantiation of the template `T`. Since the required type `U` is not concretized, its requirement *propagates* to `T2`. Also note that while the class `C` in the example above is defined directly within `T2`, it could just as well have been defined in `T`, or in a package outside `T2`.

In the paper, we compare and contrast PT*r* to other approaches to generic programming, specifically C++, Java and Scala.

6.6.2 Discussion

The research behind this paper started out as a case study for generic programming in PT, in particular looking at the Boost Graph Library [107] and the Apache Commons library [2]. Introducing new programming language concepts was not an initial goal for the work. As the work progressed, however, we realized that the constructs of what is in the paper referred to as *basic PT*, meaning PT as described in [12], were not optimal for programming heavily parameterized generic libraries such as the Boost Graph Library. That is not to say that they were not applicable or indeed able to fully express the genericity of the existing programs we studied, however, we realized that there was considerable room for improvement. In particular, we wanted to a larger degree to be able to adapt generic specifications, and we wanted to limit unnecessary repetition of generic declarations. This lead to the development of PT*r*.

Below, we discuss some issues that were not treated in any detail in the paper.

Implicit vs explicit concept maps. As mentioned previously in this thesis, a *concept* is, in the context of generic programming, a set of requirements consisting of required operations (methods) and data type constraints.

A *concept map* is a mapping from a concept to an implementing type or a set of implementing types. A concept map may be implicit, which means that an implementation of a concept is automatically inferred by the compiler for fitting types, or a it may be explicit, in the form of a declaration that explicitly states that one or more types implements a given concept, and furthermore declares how this implementation fulfills the requirements posed by the concept (e.g. if the concept and an implementing class have different names for the same operation, a concept map can state that they are indeed the same).

In PT*r*, the mapping of concrete types to concepts is done explicitly as part of an instantiation. Stroustrup has argued that for a potential future version of C++ supporting concepts, mapping of concepts should be automatic: *"concept maps must be implicit, classes that 'obviously match' concepts must match"* [122, p. 23].

If one were to support this in a version of PT for Java, the main challenge would be that instantiations in packages could leave required types that were not concretized. Thus, we would have to introduce a form of structural subtyping as well, in the vein of e.g. Whiteoak [56], where for instance method parameters can have structural, as opposed to purely nominal, constraints. Required types could then, potentially, be unified with structural types.

Traditional type parameters to templates. As briefly described in preceding papers, PT supports "traditional" type parameters to templates, in the sense that they resemble the type parameters of Java, at the template level. This allows the developer to collectively parameterize a set of template classes. The functionality of such type parameters is subsumed by the constructs introduced in PT*r*, and the "traditional" style of type parameters are, in the context of this thesis, thus considered obsolete. Note, however, that this only applies to parameters to *templates*; Java-style type parameters for *classes* are still supported (and have no direct equivalent in required types, which are intended for a different purpose).

Template parameters to templates and required types. Paper VI does not explicitly deal with template parameters as introduced in Paper V, mainly due to space constraints imposed by the publication venue. However, templates as parameters and required types are orthogonal features, and since required types propagate like other template types, they work well together. For instance, consider the expression problem example from Section 12.3.7 of Paper V (corresponding to Figure 6.2), with an added required type for some kind of additional data attached to the expressions:

```
template Expressions {
   required type Data { ... }
   abstract class Exp { }
   class Plus extends Exp { Exp left, right; Data d; }
   class Num extends Exp { int value; Data d; }
}
```

The required type `Data` is used in the `Plus` and `Exp` classes above. Below, we see the `MultExpressions` template, which adds a new class `Mult` to the set of expression classes, and utilizes template parameters to support flexible composition of "extensions" to the basic `Expressions` template. This new class `Mult` can utilize the required type `Data` directly, since it *propagates* from the `Expressions` template by way of the `E` parameter.

```
template MultExpressions<template E inst Expressions> inst E {
   class Mult extends Exp { Exp left, right; Data d; }
}
```

Below, we see the package `CombinedExpressions`, which instantiates the `Expressions` template along with several "extensions" nested within one another by using parameterized templates. The `ValueExpressions` and `PrintExpressions` are not shown here, but they follow the same general approach as the `MultExpressions` template above, adding capabilities for calculating the value and printing expressions, respectively. The example below is as such equivalent to the corresponding example in Section 12.3.7 (and the one in Figure 6.2), except that the new required type `Data` is concretized to `String`, and the `d` field of the `Mult` class is printed (which is safe because at this point we know it is a `String`). The concretization is now mandatory, since `CombinedExpressions` is a package.

```
package CombinedExpressions {
   inst MultExpressions<ValueExpressions<
      PrintExpressions<Expressions>>> with Data <= String ;

   class Mult adds {
      void print() {
        out("("); left.print(); out("*");
        right.print(); out(")"); out("Data:" + d);
      }
      int value() { return left.value() * right.value(); }
   }
}
```

Note that required types may also be concretized for an actual given parameter, i.e. between the angled brackets (< and >), in an instantiation.

Access modifiers. While not explicitly mentioned in paper VI, it seems clear that a required type declaration must be implicitly public; it makes no sense to declare a required type that cannot be seen (and thus not concretized) from outside the template. Correspondingly, the structural method signatures declared in a required type must have a public match in a type that is supplied as a concretization for the required type.

6.7 Paper VII: *Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code*

Authors. Eyvind W. Axelsen and Stein Krogdahl. Axelsen is the main author, and was responsible for writing all the text, and designing the transformations. The idea and main approach was developed jointly by Krogdahl and Axelsen.

Publication. Published in Proceedings of the 11th International Conference on Generative Programming and Component Engineering, 2012 (GPCE'12) [9]. Acceptance rate 43%.

6.7.1 Summary

In this paper we aim to give a more precise definition of the semantics of the PT language than what has been done in previous papers. To achieve this, we present a series of source-to-source transformations defined for a core subset of PT for Java. This subset of PT is called *core PT*, or C-PT for short. Programs in C-PT can be translated to plain Java code by using the transformations defined in the paper. C-PT includes instantiations, addition classes, overrides in both the subclass dimension and the addition dimension, and class merging. Thus, in our opinion, it captures the essence of full PT.

Using the transformation we *define* the meaning (or semantics) of a C-PT program directly as that of the Java program produced by the transformation.

The transformation to Java has been designed to support the following desirable properties:

- A correct PT program will always end up as a correct Java program.
- The transformation preserves the semantic bindings of names to declarations from the PT code, as described more intuitively in previous papers. This is somewhat challenging to achieve because the transformation composes independently developed code, and the output of the transformation is to be compiled by a standard Java compiler, which will perform name resolution and binding independently of any PT-specific semantics.
- Externally visible names (names that might be referred to in client code) are preserved through the transformation unless renamed by the programmer of the client code.
- The resulting Java program is efficient at runtime; the use of the transformation does not introduce any significant runtime overhead.

The paper presents a transformation consisting of four steps (or sub-transformations) that are executed sequentially to transform C-PT code to Java code. This four-step transformation is applied repeatedly, for each instantiation, in a certain

order, until all the instantiations have been processed. The end result, when all the instantiations have been processed, is an ordinary Java package.

The individual transformation steps can be summarized as follows:

- **The fortifying transformation step** makes a template more resilient to unintentional rebinding when involved in further composition and adaption, by explicitly qualifying references to methods and fields, and by introducing fresh names for local declarations.
- **The renaming transformation step** performs renaming of program elements (classes, methods, fields) in a template according to the corresponding `inst` statement.
- **The addition-handling transformation step** prepares individual instantiated templates for composition with the instantiating template/package, taking addition classes into account.
- **The composing transformation step** performs the actual composition of the classes from the instantiated templates with the addition classes of the instantiating template/package. Since the previous transformations have already prepared the instantiated templates for this step, the composing transformation's job is mainly to textually combine pieces of code.

We can view a PT program as a tree of instantiations, where an instantiation corresponds to an edge and an instantiated template or a package to a node in the tree. The root node will thus be a package, and the leaf nodes are templates that do not instantiate any other templates. We can then iteratively apply the transformation, with its four steps, to the templates, working from the leaf nodes and in towards the root.

For each of the four transformation steps there is a corresponding lemma (with a proof sketch/outline), based on which we can conclude that the result of the transformation will be a legal Java package.

6.7.2 Discussion

There are a few important aspects of full PT that are not treated in any detail in the paper, primarily due to space constraints enforced by the publication venue. Below, we discuss some of these aspect in more detail, focussing on how they could be transformed to plain Java using the same approach as in the paper (however, we will not attempt to define any additional lemmas for the transformation steps).

Calls to overridden template class methods by using `tsuper`. The `tsuper` construct, introduced and described in Section 12.3.2 of Paper V, is used to call template methods that are overridden in an addition class. For instance, if a method `void m()` is declared in a class `C`, and an addition class `C'` to `C` also defines a method `void m()` (i.e., a method with an override-equivalent signature [58, sec. 8.4.2]), the former method can be called using the `tsuper.m()` syntax. However, we did not find room

in Paper VII to include a treatment of the `tsuper` construct, but we will discuss it briefly below.

An important point in Paper VII is that we reuse the lookup and binding semantics from Java to analyze/check the code inside templates, and then utilize this to perform our own analysis and transformation.[4] This allows us to define the semantics of PT for Java relative to standard Java semantics, but it also consequently entails that all the code that we want to analyze/check in this manner needs to follow Java's rules for syntax and semantics. Thus, for constructs that are *not* part of standard Java, we need to perform a preliminary *syntax-driven* transformation of PT code to Java code. In other words, we cannot directly rely on semantic bindings to transform `tsuper` calls to Java code.

Instead, we will syntactically transform `tsuper` calls to ordinary method calls by, basically, transforming `tsuper.m(...)` calls to ordinary method calls of the form `$tsuper_m(...)`, and create the corresponding method signatures. A problem with this approach is that without semantic bindings, we cannot really know which overridden method a `tsuper` call refers to, if there are multiple candidates. In order to deal with this, we find the set of methods that the call *might* be bound to, which is the set of methods with the same name and the same number of parameters as the call. This can be resolved purely syntactically, and we then mark these methods with standard Java annotations, so that we can return to them after the semantic analysis has been performed for further transformation.

Before we describe the steps involved in transforming code containing `tsuper` calls in detail, we will consider an example. Below is a PT program skeleton, where fragments in the form of `<Block #N>` are placeholders for blocks of ordinary Java code:

```
template T {
   class C {
      void m(String s) { <Block #1> }
      void m(Object o) { <Block #2> }
      void m(int i)    { <Block #3> }
   }
}
template U {
   inst T;
   class C adds {
      void m(String s) { <Block #4> }
      void m(Object o) { <Block #5> tsuper.m(x); <Block #6> }
   }  // x in the call to tsuper.m above can be any expression
}
```

The template `T` above is what is in Paper VII referred to as a *closed template*, that is, a template that does not instantiate any other templates, and whose contents are defined

[4] In an implementation, we can thus reuse any standard Java compiler, as long as it can expose semantic bindings, for instance in the form of an abstract syntax tree.

in a true subset of the Java language. The template `U`, on the other hand, is an *open template*, meaning that it contains `inst` statements and potentially other PT-specific constructs, like `tsuper` calls and addition classes. As we can see from the example, we cannot, without a semantic analysis of both `T` and `U`, know to which overload of `m` the `tsuper` call in the addition class refers.

In order to deal with this in a syntactic transformation, we will utilize placeholder methods (method *stubs*) and annotations that allow us to navigate the original structure when the semantic bindings are in place. We will use the following annotations:

- `@MethodId(int)` — This annotation is used to assign a unique integer identifier to methods for which we will introduce an additional "companion method", implemented as a method stub that only throws an exception. Its name will be the name of its "companion" prefixed with "`$tsuper_`".
- `@RefersTo(int)` — This annotation is used on the newly generated method stub to link it to the method identified with the `@MethodId` annotation described above.
- `@TOverridden` — This annotation is introduced in Paper VII, and is used to mark template class methods that are overridden in addition classes. Such methods can be identified syntactically by just considering their signatures in isolation, since Java does not allow contra-variant parameter types in method overrides (and we have explicitly excluded covariant return types from C-PT).

The code below shows an intermediate result from an extended version of the addition-handling transformation step from the paper (the transformation step is described in detail below). In the code, each potential target for the `tsuper.m(x)` call is generated as a new method stub with an identical signature that (for now) just throws an exception, with annotations as described in the list above.[5]

```
template T {
   class C {
      @MethodId(1) @TOverridden void m(String s) { <Block #1> }
      @MethodId(2) @TOverridden void m(Object o) { <Block #2> }
      @MethodId(3)               void m(int i)    { <Block #3> }
      @RefersTo(1) void $tsuper_m(String s)
         { throw new Exception("This is just a placeholder"); }
      @RefersTo(2) void $tsuper_m(Object s)
         { throw new Exception("This is just a placeholder"); }
      @RefersTo(3) void $tsuper_m(int i)
         { throw new Exception("This is just a placeholder"); }
} }
```

[5]The reason for throwing an exception instead of just having an empty method stub, is to appease the compiler/semantic analyzer in case the method is declared to return a value. Since the methods are declared as `void` in this example, the method body could just as well be empty, but for simplicity we will throw an exception from all method stubs that we generate in this manner.

```
template U {
   inst T;
   class C adds {
      void m(String s) { <Block #4> }
      void m(Object o) { <Block #5> $tsuper_m(x); <Block #6> }
   }
}
```

The composing transformation step will then, subsequently, concatenate the templates `T` and `U`. Methods that have a `@TOverridden` annotation are superfluous, and can be removed in this step, but their *bodies* must be preserved in the newly generated method with the "`$tsuper_`" prefix (which can easily be navigated to using the `@RefersTo` annotation). The template `U` will then look like the following:

```
template U {
   class C {
      @MethodId(3) void m(int i) { $tsuper_m(i); }
      @RefersTo(1) void $tsuper_m(String s) { <Block #1> }
      @RefersTo(2) void $tsuper_m(Object s) { <Block #2> }
      @RefersTo(3) void $tsuper_m(int i) {  <Block #3> }
      void m(String s) { <Block #4> }
      void m(Object o) { <Block #5> $tsuper_m(x); <Block #6> }
   }
}
```

Now, we again have a closed template, and the contents of `U` are thus in a subset of the Java language. We can now utilize standard Java lookup rules and establish the semantic bindings for all the code in `U`. If there are any generated methods (i.e., methods with a `@RefersTo` annotation) that are never called, they can safely be removed. All the annotations can subsequently be removed.

Amending the transformation steps from Paper VII. We shall now leave the example above, and consider the general case.

The addition-handling transformation step, which is described in more detail in the paper in Section 14.4.3, is extended as shown below, in order to handle `tsuper` calls. The discussion below is somewhat technical, and relates directly to the approach in Paper VII, so having read that paper in advance is recommended.

Below, steps 1 and 4 are identical to steps 1 and 2 as presented in the paper, respectively, while steps 2 and 3 are new:

For each class `C` in an instantiated template `T`:

1. If a variable[6] named `v` is declared in `C`, and a variable named `v` is also declared in an addition class to `C`, then the variable named `v` in `C` should be given a new fresh name, and every usage of `v` in `T` should be updated to the new fresh name.

[6] We know that it is an instance variable since we disallow static declarations in C-PT.

2. If there are any names in `C` that already are prefixed with `$tsuper_`, add another `$tsuper_` prefix in front of that, and update references accordingly. This is done in order to make sure that the next step does not introduce any ambiguity.

3. If an addition class `C'` to `C` contains a call to a method `m` qualified with `tsuper`, and with parameters ap_1, ..., ap_N, $N \in \mathbb{N}_0$ (i.e. the call is `tsuper.m(`ap_1`, ..., `ap_N`)`), then do the following for each such call:

   - For every method `m'` with the same name as `m` and with N formal parameters in `C`, create a copy of the signature of `m'`, and give the copied method signature the prefix `$tsuper_` to its name. Furthermore, add an annotation to the method declaration of `m'`, annotating it with `@MethodId(k)`, where `k` is generated by the integer number generator. Annotate the new method's declaration with a `@RefersTo(k)` annotation, where the `k` is the same as for the `@MethodId` annotation. The body of the new method should contain only the following code:

     ```
     throw new Exception("This is just a placeholder");
     ```

     The reason for creating an extra method instead of only renaming the existing method is to account for virtual calls.
   - Change the call `tsuper.m(`ap_1`, ..., `ap_N`)` to `$tsuper_m(`ap_1`, ..., `ap_N`)`

4. If a method named `m` is declared in `C`, and an addition class to `C` also defines a method named `m` with identical signature, then mark the method in `C` with a `@TOverridden` annotation.

We have now made the addition-handling transformation step capable of dealing with `tsuper`, but we also need to make a small adjustment to the composing transformation step (originally defined in Section 14.4.4). Most of the transformation step is identical to the one presented in the paper, but we need to make a small change to Step 1 (a). Here, we will copy the body of a `@TOverridden` method to its corresponding "`$tsuper_`" stub.

1. For every template $T_i \in S$, for every class $C_i \in T_i$:

   (a) If C_i has a method marked with a `@TOverridden` annotation, this method should also have a `@MethodId` annotation. Find the method with the corresponding `@RefersTo` annotation, and move the body of the method with the `@TOverridden` to the body of this method, replacing any existing statements. Then delete the method marked with `@TOverridden`, and remove the `@RefersTo` annotation from the other method.

   (b) If an addition class with the same name as C_i does not exist in U, create an empty addition class with this name in U.

(c) Copy the body of C_i and append it to the body of the addition class with the same name in U.

2. For every addition class $A_i \in U$, transform the addition class to an ordinary class (i.e., remove "`adds`"), handle `extends` clauses, etc.

3. For every `inst`-statement $I_i \in U$, remove I_i.

Finally, we need to introduce one additional transformation step for handling the remaining annotations in the code. We may call it the *tsuper transformation step*, and this step depends on the semantic bindings from the composed template code, which can be established after the composing transformation step, as described above, is completed:

1. For every method call that is bound to a method with a `@RefersTo(k)` annotation, find the corresponding method with a `@MethodId(k)` annotation. Move the entire body of the latter method to the former, and insert a call to the former in the latter, utilizing the formal parameters as arguments in the call. If the method is not declared to return `void`, prefix the call with a `return` statement. Remove the `@MethodId` and `@RefersTo` annotations from the two methods.

2. If there are any methods with a `@RefersTo(k)` annotation that are never called, delete them. If a corresponding method marked with `MethodId(k)` exists, remove the annotation from this method.

3. If there are any methods that start with the prefix `$tsuper_` that are never called, remove these methods.

With this final transformation step, and the amendments to the addition-handling and composing transformation steps, we can now transform C-PT code that includes `tsuper` calls to ordinary Java code.

Constructors. Constructors in template classes will be represented as (i.e., transformed to) ordinary methods (with mangled names) when the template is instantiated. The handling of `tsuper(...)` constructor calls has thus much in common with the handling of `tsuper.m(...)` method calls as described above. This way of doing it has one important drawback, as mentioned in the paper, and that is that `final` variables cannot be assigned in template class constructors.

As described in Section 6.5.2, the current PT implementation now supports calls to template class constructors using `new`, with any number of parameters. A constructor signature with an `assumed` modifier must then be declared, in order to state that the final package class *must* implement a corresponding real constructor. To allow a closed template (i.e. a template without `inst` statements or other PT-specific constructs) with assumed constructors to be type checked and semantically bound using ordinary Java rules, any assumed constructors must be transformed into ordinary declarations, and we do this using an `@Assumed` annotation according to the following rules:

For every `assumed` constructor signature declaration a in every template class C:

1. If an ordinary constructor with the same signature as *a* exists in *C*, annotate the existing ordinary constructor with an `@Assumed` annotation
2. If, on the other hand, an ordinary constructor with the same signature as *a* does not exist in *C*, then create a new ordinary constructor with the same signature as *a* and an empty body, and add the `@Assumed` annotation to the new constructor.
3. Remove the declaration of *a*.

The addition-handling transformation step from Section 14.4.3 must be extended with an additional step to handle calls to template class constructors using `tsuper(...)`. This step will be very similar to the step we added for `tsuper.m(...)` calls above, however, we need not create an extra method; giving the constructor a unique normal method name, will suffice. Calls to `tsuper(...)` obviously need to be changed accordingly, as will calls to `this(...)`.

We also need to add corresponding handling of constructors to the tsuper transformation, as mentioned for method calls above.

Finally, we need to add a check that ensures that, the final addition class (in a package), all assumed constructors are declared.

Interaction with ordinary Java generics. As mentioned briefly in the paper, in order to support ordinary Java-style generics in template classes, we need to impose some restrictions, and perform some extra checks, specifically tailored to the semantics of the Java language. However, these restrictions and checks are not described in the paper, so we will take a closer look here.

As an overall principle, we state that Java-style generic class declarations are allowed in PT, as long as the restrictions below are upheld.

The first restriction has to do with merging, and as mentioned in Paper VII, only classes with the exact same number and bounds of type parameters can be merged. We also need to restrict renaming so that one cannot rename a class to the name of a declared type parameter (template-wide).

Furthermore, we need to disallow overloads in an addition class where the only discriminator for overload resolution is one or more generic types. That is, if a method named `m` in a template class `C` has formal arguments of type $P_1, ..., P_N$, an addition class cannot add a new method named `m` with parameter types $Q_1, ..., Q_N$ if, for every $1 <= i <= N$ we have that $P_i = Q_i$ or either P_i or Q_i is a generic type parameter, and there is at least one generic type parameter such that $P_i \neq Q_i$.

To see why, we consider the following example:

```
template T {
   class A {
      void <K> m(K t) { ... }

      void f() { m("Test"); }
   }
}
```

Note how, in template `T`, the call to `m` in `f` will bind to the single implementation of `m` available. However, now what if one were to create the following template with an addition class for `A`:

```
template U {
   inst T;

   class A adds {
      void m(String s) { ... }
   }
}
```

Since we want to preserve the semantics of `A` in `T`, so that the call to `m` in `f` resolves to the method in `T.A` instead of the new version in `U.A`, we could be tempted to modify the fortifying transformation from Section 14.4.1, and transform the call in `T` to be explicitly generic, e.g. "`m<String>("Test")`". While this seems intuitively correct, the Java specification, surprisingly, states that the overload resolution mechanism might choose a non-generic method implementation even for a call-site that explicitly specifies generic parameters [58, sec. 15.12.2.1].

Thus, in order to maintain static semantics in the presence of overloads with generic parameters, we need to disallow such overloads when the only difference in method signatures is the generic parameters. Clearly, this restriction stems from Java idiosyncrasies rather than being directly related to the PT mechanism per se, and if we had chosen e.g. C# as our target language instead of Java, this particular restriction could be lifted (however, we might find ourselves in a similar situation in other areas, due to the idiosyncrasies of C#).

Chapter 7

Concluding Remarks and Future Work

This chapter concludes Part I of this thesis. It has two parts. The first (7.1) is a discussion on how, or to what extent, we have reached our goals. The second part (7.2) briefly discusses directions for future work.

7.1 Have We Reached Our Goals?

In this section, we discuss to what extent the goals and desiderata presented in the introduction and in Chapter 4, respectively, have been met by this thesis work.

We start, in Sections 7.1.1 through 7.1.5 below, by considering the desiderata from Chapter 4, and try to give a verdict for the degree of fulfillment that we have achieved for each of them.

7.1.1 General Desiderata

Desideratum 1: ***Applicability to more than one language.*** The core constructs of the mechanism should be designed so that they can be added to a variety of object-oriented programming languages.

Verdict. The PT mechanism has, in this thesis, been successfully applied to the object-oriented programming languages Java and Groovy. Prototype implementations have been developed, and examples have been programmed utilizing these implementations. There is also a master's thesis by Stordahl [118] that applies the mechanism to the Boo.NET language; for this thesis Krogdahl and Axelsen had the roles of main and co-supervisor, respectively.

The three languages are quite different (within the realm of object-oriented languages), which in itself provides some support for the stance that PT is applicable to a wide array of languages. However, for a more definite answer to this desideratum, it seems that more studies are required.

Desideratum 2: ***Low performance overhead.*** The runtime performance overhead of applying the mechanism should be acceptable for use in real-world scenarios.

Verdict. For the version of the mechanism that is applied to Java, using a transformation-based approach similar to the one outlined in Paper VII, the runtime overhead of the mechanism is virtually non-existent, save for a few cases where the overhead of an extra method call is incurred. For some scenarios related to generic programming, compile-time specialization of the code will actually lead to code that is a little faster than the corresponding plain Java code (because the number of casts, that would otherwise be inserted by the Java compiler, is reduced).

An alternative to the transformation-based, compile-time specialization approach of Paper VII would be to generate code for each template only once (as opposed to once *per instantiation*). Such an approach could typically lead to smaller programs in terms of compiled code size, but it would probably require additional runtime support in terms of lookup tables etc., incurring additional runtime performance overhead.

For the Groovy version of the mechanism, there is a quite substantial overhead involved with the runtime instantiation of a template. Whether or not this overhead is acceptable is difficult to answer on a general basis, but it is clear that a lot more work can be done in order to improve the performance of the current implementation.

There is no overhead involved compared to ordinary Groovy code when using objects created from instantiated template classes (e.g. when making method calls or accessing instance variables) in the current implementation.

If user-defined strategies are applied to runtime instantiations in Groovy PT, there is another performance hit when the instantiation is performed, and the overall performance of such instantiations will obviously depend on the actual code written by the programmer for the strategy.

Desideratum 3: ***Applicability to real-world problems.*** There should be a battery of examples that show how the mechanism can be applied to, and be useful for, real-world problems and programs.

Verdict. The papers in this thesis, as well as the discussion in Chapter 6, are all to a certain extent driven by examples, which in themselves are intended to provide arguments for the utility of the mechanism. For instance, we have implemented all the design patterns from [53], as well as the Active Object pattern [80]. We also provide an implementation of a non-trivial subset of the Boost Graph Library [107], as well as a novel solution to the Expression problem [134].

While more examples and case studies are always a good thing, and true real-world experience is hard to attain in software research projects, we find it reasonable to say that the work in this thesis at least strengthens the stance that the mechanism is useful also for real-world problems, on the basis of the examples and prototype implementations provided by this thesis.

Desideratum 4: ***Specifications.*** There should be specifications that describe the syntax and semantics of the mechanism, in the context of specific, real-world, target languages.

Verdict. A specification in the form of a transformation to Java for a core subset of the mechanism as applied to Java is presented in Paper VII. Some of the constructs that were left out of Paper VII are discussed in Section 6.7.2. A specification for the full mechanism, preferably in the style of the Java Language Specification [58], would nevertheless have been desirable. Work is in progress with regard to creating such a specification.

For the mechanism as applied to Groovy, we did not find time to write a specification. However, since the mechanism is implemented as an internal DSL in Groovy itself, the implementation is much less of a black box compared to the Java version, and the semantics are thus more transparent. Still, an API specification for the classes, properties, and methods that make up the DSL would have been desirable.

7.1.2 Separation of Concerns, Modularization and Composition

Desideratum 5: ***Semantics preservation with respect to composition.*** The composition operations should preserve the semantics of the composed elements.

Verdict. In Paper VII, we show how a translation to Java preserves the semantics of composed code in PT for Java.

For the dynamic variant of PT, the semantic bindings of individual components cannot readily be established (without actually running the program), amongst other things because the language supports multiple dispatch (based on the runtime type of actual parameters), a meta-object protocol, and runtime addition of methods.

Desideratum 6: ***Composition of previously unrelated components.*** The mechanism should support composition of elements that were not explicitly developed with their mutual composition in mind.

Verdict. Templates in PT with no prior relationship can be combined by instantiating them in the same template or package. Furthermore, classes within such templates can be merged to form new classes, without any prior relationships, and without requiring multiple inheritance in the target language.

However, to avoid introducing multiple inheritance we must introduce a requirement that superclasses to merged classes must also be merged, which in certain scenarios might be a restriction that limits the possibilities for taking full advantage of combining independently developed classes.

For the statically typed version of the language, generic constraints in the form of required types are important in order to express requirements for a composition without tight coupling to other elements. Required types may also be merged, thus supporting equality constraints for previously unrelated generic concepts.

Furthermore, the support for a simple subset of AOP constructs in the form of pointcut and advice can aid the programmer in expressing loosely coupled requirements, through the support for declaring abstract pointcuts that express that code in

the module will react when *something* of interest happens, yet what this something is must be defined by another, more specific, module.

Desideratum 7: ***Isolated adaption and composition.*** Changes to, and compositions involving, an element should be isolated from the rest of the program.

Verdict. In the statically typed version of PT, the effects (on the static semantics) of an instantiated template, including any adaptions, are scoped to the instantiating template/package. Also, the AOP constructs introduced in Paper I can only refer to declarations within a template's boundaries. Thus, unrelated parts of the program are not affected.

In the dynamically typed version, the effects of an instantiation are scoped to the variable to which the result is assigned, thereby allowing all the mechanisms for encapsulation in the Groovy language to also apply to instantiations.

Somewhat related to this topic is the issue of access modifiers for declarations in templates (for both statically and dynamically typed variants). An approach to this is outlined and briefly discussed in Paper V, however, it seems clear that further research is needed in this area in order to finalize the design of this in PT.

7.1.3 Retroactive Adaption, Composition and Implementation

Desideratum 8: ***Name changes of program elements.*** It should be possible to modify the names of the program elements in a composition so that new names replace all occurrences of the old names (based on established bindings, cf. Desideratum 5) in the composed result.

Verdict. The statically typed version of the mechanism fulfills this desideratum, as interfaces, classes, methods, and fields can be renamed. Furthermore, it also supports renaming of generic constraints in the form of required types, and structural signatures within such constraints.

However, it should be noted that the support for renaming leaves something to be desired in terms of debugging and error message support. It can in many cases be hard to trace errors back to the original source code as written by the programmer, especially since we partially base our support for semantics preservation on name mangling of non-public attributes.

The dynamically typed version of the mechanism, through its implementation in Groovy, supports renaming of classes. We *chose* not to support renaming of methods and fields, since it in general is not possible to rename every referring use-site in a safe manner (due to the dynamic nature of the Groovy language). An alternative approach could be to allow the programmer to specify *aliases* instead of renames, similar to what is done for Traits.

Desideratum 9: ***Refinement at any level.*** It should be possible to retroactively refine existing definitions at any level in an inheritance hierarchy, i.e., not just at the leaf nodes.

Verdict. This is supported by addition classes, addition interfaces, and additions to generic constraints in the form of required types. These constructs support pure additions and overrides. Such refinements can be made to elements at any level in an inheritance hierarchy. Refinement in the form of renaming, composition, and interface implementation is also supported at any level, as detailed in other desiderata.

Desideratum 10: ***Retroactive interface implementation.*** It should be possible to state that existing classes implement existing interfaces, without having to change the source code of either.

Verdict. This is supported by the `adds implements` clause of an addition class declaration, and further aided by the possibilities for renaming (cf. Desideratum 8) in order to make a class conform to an interface. Also, the fact that required interfaces (from the work on required types) can be adapted and made to fit existing interfaces, which template classes can then implement, works towards fulfilling this desideratum.

7.1.4 Module Adaption and Composition in Dynamic Languages

Desideratum 11: ***Runtime composition and adaption.*** It should be possible to perform composition and adaption of program elements at runtime, based on runtime criteria.

Verdict. This is supported by the implementation of Groovy PT, for which the instantiation can be tailored based on runtime conditions in the program.

Desideratum 12: ***Meta-level control over composition.*** It should be possible for the programmer to control the composition of elements through a meta-level protocol.

Verdict. Meta-level strategies written by the programer can be applied to any instantiation in Groovy PT. Control over e.g. conflict resolution, name changes, precedence, etc. is available. Furthermore, the programmer of a module can equip that module with meta-level capabilities so that it can interact with its own instantiation at runtime.

7.1.5 Generic Programming and Adaption

Desideratum 13: ***Multi-type concepts and constraints.*** It should be possible to express non-trivial generic concepts in terms of a set of related classes, and it should correspondingly be possible to parameterize multiple classes with a single generic parameter.

Verdict. With required type definitions in PT, concepts and constraints that span multiple types (within one template) can naturally be expressed.

For the dynamically typed version of PT, we have not added any support for generics. This is partly due to the fact that generic constructs have much less utility in a language without static typing. Expressing requirements explicitly may still have value, though, and while this can be partly attained through abstract classes and interfaces, a more direct approach could perhaps be valuable.

In [17], the authors present an approach to adding generics to the (dynamic) Smalltalk dialect Pharo, focussing on object creation. Like for Groovy, references to classes in object creation statements are typically resolved at compile-time, and thus there is a certain potential for the utility of generic implementations that abstract over the classes from which objects are created. However, with Groovy PT, a similar effect can be achieved through runtime template instantiations.

Desideratum 14: ***Propagation of concepts and constraints.*** It should be possible to specify generic concepts and constraints that propagate with module composition.

Verdict. Required type definitions naturally propagate to instantiating templates. Selective concretization of required types allows the programmer to bind (potentially propagated) required types at the level where it is appropriate, and thus prevent further propagation.

7.1.6 Overall Goals

In Section 1.1, we formulated two rather broad, overall, research goals:

1. **Formulate a solid and useful design for the Package Template mechanism**. This involves carrying out research to explore the design space from the initial ideas of [76], and to experiment with and assess features in light of experience gained through programming with PT. Furthermore, it entails to make explicit and solidify the fundamentals of the mechanism.

2. **Apply the mechanism to real-world languages, patterns and problems**. This involves coming up with implementations and specifications that relate to specific real-world languages, to formulate existing patterns and practices in terms of our mechanism, and to utilize this in order to solve real problems.

These goals are addressed by this thesis in the following way:

- **For goal number 1**, the desiderata discussed above in Sections 7.1.2 through 7.1.5 address the main points from this goal. We have extended the original suggestions for a mechanism like PT with several new constructs that allow the expression of a wider range of reusable class libraries in the form of templates. The utility of the features is demonstrated through examples. We have also defined the semantics of the core constructs of the mechanism.

- **For goal number 2**, the general desiderata in Section 7.1.1, as well as the artefacts presented in Section 1.3, address the application of the PT mechanism to real-world languages (Java and Groovy), design patterns, and example problems (such as the Expression problem and the Boost Graph Library).

Thus we find that we can conclude that even though some individual desiderata are not accomplished in full, the overall goals of this thesis have been reached to a considerable extent.

7.1.7 Weaknesses

Even though we, with some exceptions, have reached the overall goals for this thesis as presented in the introduction, we have still identified areas that challenge the validity of the claims made in this thesis. The following two areas do, in our opinion, represent the main weaknesses.

Lack of formalization. The work with this thesis has largely been example-driven and exploratory, and the main focus has not been on a rigid formal approach. Even though both papers V and VII contain pieces of formalization, it seems clear that a full formalization of the mechanism would be beneficial for the work, and better substantiate our claims of type safety and semantics preservation.

Limited set of examples. The time allotted for working with a PhD thesis is limited, and even though there is a quite substantial amount of implemented code that accompanies this thesis in the form of prototypes, examples, and case studies, it is not enough to fully grasp the *real-world* applicability of the mechanism, since none of the examples are utilized in a real, industrial, context.

Other researchers and master students in the SWAT project have also done some programming with the PT mechanism, implementing various libraries and programs, but still, the combined PT programming experience in the group is limited.

Thus, the utility of the mechanism for *programming in the large* is still pretty much an unproven point. Industrial application of research-level concepts and constructs is not an easily attainable goal, but it would doubtlessly strengthen the propositions in this thesis.

7.2 Future Work

When considering topics for future work, a natural target would be to try to amend the main weaknesses addressed in Section 7.1.7. Below, we briefly discuss these two topics before we move on to other potential directions for future work.

Formalization. A formalization of the core PT constructs in the style of e.g. Featherweight Java [73] would presumably be a good way to approach a more rigorous formal definition of a core subset of the mechanism, in order to prove this core subset to be type safe. Much of the work involved would probably be centered around adapting the definition of the class table in Featherweight Java to understand the PT `inst` statement and the classes it "produces" relative to the defined templates.

Case studies. Larger and more realistic case studies would clearly improve confidence in the mechanism and its claims to utility for reuse and adaption of class libraries. Performing such case studies would more or less require a compiler/runtime-system that is more robust than the prototype variants we have today.

The lack of proper programming experience with the mechanism is perhaps most notable for the dynamic version of the language, so it could be tempting to start with that variant. For instance, it could be interesting to investigate the aptitude of package templates on existing popular libraries/frameworks such as e.g. Grails [115] or Gaelyk [79].

Correspondingly, for the statically typed variant of PT, it would be interesting to try to rewrite libraries/frameworks of a certain size to see what can be gained in terms of modularity and reuse of code by applying the constructs introduced in this thesis. Examples that could be well suited for such a reimplementation include the full Boost Graph Library [107] (for which a subset was implemented in Paper VI), the Java Swing GUI library [82] (or, perhaps more realistically, a subset of this), and various components in the Apache Commons library [2].

Unification of templates and packages. A question that we have asked ourselves repeatedly throughout the process of developing the constructs presented in this thesis is: is the sharp distinction between templates and packages really necessary?

It seems that it would be useful if any package could be used as a template, and vice versa. I.e., it would probably be a benefit in having these two concepts be *one and the same*. As PT stands today, there are a few limitations that need to be resolved for this to work. To begin with, the current scheme for constructor implementation relies on distinguishing template classes and package classes, in order to enforce rules for calling super and tsuper constructors.

Furthermore, PT enforces a restriction that there can be no cyclic template instantiations. On the other hand, cyclic package references are not uncommon at all, and this is what we resort to for PT as well if cycles are required: instantiate one template in a package, and reference this package from the other template (which will lead to cyclic package references when the template is instantiated in another package). For a unification of the two concepts, another resolution to this issue must be found.

Template instance parameters. When templates are instantiated in PT, the rules of the mechanism ensure that there will be no diamond structures in the resulting template instance graph. In fact, in a program, the resulting instances will always form

a tree, with a package at the root. However, diamond-like structures of template instances can clearly be useful for e.g. specifying several independent extensions to a common base template, and then combining a selection of such in another template. The template parameter approach introduced in Paper V, and further defined in Section 6.5.2, achieves some of this, but ideally we would like an even more flexible solution.

Such a solution should allow the programmer of a parameterized template to specify that some of its parameters should share a common instance of another template, while still leave open the possibility that some of the involved instances, in later instantiations, can be shared with yet other templates that might be unknown at the time when all the involved templates (at that point in time) are written.

Wider array of adaptions of template classes. Since each template instantiation is independent from any other instantiation, the degree of freedom to adapt the template contents is large. Thus, it is possible to support much more drastic operations than what PT currently supports (especially relevant for the statically typed version), such as e.g. merging of variables, a wider range of aspect-programming constructs and capabilities, constructor combination schemes, deletion of attributes from classes, or, for that matter, deletion of entire classes (probably provided that there would be no dangling references to them as a consequence of the deletion). Furthermore, class morphing, e.g. something analogous to what is supported by the compile-time reflective capabilities of MorphJ [71], could be an interesting and useful addition to PT.

Better implementations. There are several things that can be improved with regard to the prototype implementations that we currently have, both for the statically typed and the dynamic variant. An obvious area for improvement is to make industrial strength implementations, which would provide the opportunity to use the PT mechanism for real world programming.

Another issue is the support for debugging and error handling. The current implementation of JPT is based on transforming PT source code to plain Java. Because of this, error messages can be hard to understand, since the original structure (with templates, template classes, addition classes, etc.) as written by the programmer no longer exists. To rectify this, one would somehow have to keep track of the relationship between the original source and the generated source. There are tools that can help with at least some of this, like the Apache Commons Byte Code Engineering Library [2], which contains ways to set source file, line number, etc. for byte code in Java class files.

Taking a somewhat broader view, an interesting question in general is how to best report errors and debugging messages for mechanisms where source code elements can be renamed and merged (potentially multiple times) by the programmer. Should the original name be used, or the new one(s), or a combination?

A different issue is that of template packaging and distribution. The implementations discussed in this thesis rely on distribution of templates in source code form, both for the dynamic and the static variant. However, at least for the static variant,

the ability to separately compile templates seems worth investigating. Such an approach would facilitate distribution of templates independently of the programs that use them (which would otherwise have to be recompiled for any changes to take effect), in the same way as e.g. dynamic link libraries (dlls) on the Windows platform and Java archive files (jars) are often distributed today. However, this would also require some kind of system for indirection, e.g. for calls to renamed methods and to handle merged classes, etc. Such approaches are, obviously, not without runtime performance implications. An interesting question in that regard is whether one could get better runtime performance by using e.g. specialized byte code instructions for PT instead of being limited to what is available in current virtual machines.

Bibliography

[1] Mehdi Achour, Friedhelm Betz, Antony Dovgal, Nuno Lopes, Hannes Magnusson, Georg Richter, Damien Seguy, and Jakub Vrana. *PHP Manual*. The PHP Documentation Group, January 2013.

[2] Apache Software Foundation. Apache Commons Library. Available online: `http://commons.apache.org/` (accessed October 23rd 2012).

[3] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 101–112, New York, NY, USA, 2008. ACM.

[4] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Trans. Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[5] AspectJ Team. The AspectJ programming guide, 2003.

[6] Eyvind W. Axelsen and Stein Krogdahl. Groovy Package Templates: supporting reuse and runtime adaption of class hierarchies. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 15–26, New York, NY, USA, 2009. ACM.

[7] Eyvind W. Axelsen and Stein Krogdahl. Towards pluggable design patterns utilizing package templates. In Trond Aalberg, editor, *Proceedings of the Norwegian Informatics Conference (NIK'09)*, pages 97–108. Tapir Academic Publisher, 2009.

[8] Eyvind W. Axelsen and Stein Krogdahl. Adaptable generic programming with required type specifications and package templates. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 83–94, New York, NY, USA, 2012. ACM.

[9] Eyvind W. Axelsen and Stein Krogdahl. Package templates: a definition by semantics-preserving source-to-source transformations to efficient Java code. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 50–59, New York, NY, USA, 2012. ACM.

[10] Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller-Pedersen. Controlling dynamic module composition through an extensible meta-level API. In *DLS*

2010: Proceedings of the 6th symposium on Dynamic languages, New York, NY, USA, 2010. ACM.

[11] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 37–42, New York, NY, USA, 2009. ACM.

[12] Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. Challenges in the design of the package template mechanism. *Transactions on Aspect-Oriented Software Development*, 9:268–305, 2012.

[13] David Bacon, Joshua Bloch, Jeff Bogda, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration. Available online: `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html` (accessed October 30th 2012).

[14] John Barnes. *Programming in Ada95*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[15] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. pages 3–35. 2006.

[16] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proc. 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Alexandre Bergel and Lorenzo Bettini. Generics and reverse generics for Pharo. In Slimane Hammoudi, Marten van Sinderen, and José Cordeiro, editors, *ICSOFT*, pages 363–372. SciTePress, 2012.

[18] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: controlling the scope of change in Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[19] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[20] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45213-3_17.

[21] Andrew Black. Object-oriented programming — challenges for the next fifty years. Slides from a talk presented at the Department of Informatics, University of Oslo, August 25th 2011. Available online: `http://www.cs.pdx.edu/~black/presentations/O-J%20Dahl%20Talk.pdf` (accessed December 20th, 2012).

[22] Viviana Bono, Jarek Kuśmierek, and Mauro Mulatero. Magda: A new language for modularity. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 560–588. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-31057-7_25.

[23] Gilad Bracha. Generics in the Java programming language. Technical report, Sun Microsystems, Santa Clara, CA, July 2004.

[24] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[25] Gilad Bracha. Newspeak programming language draft specification version 0.07, 2011.

[26] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *OOPSLA/ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[27] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM.

[28] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In Theo D'Hondt, editor, *ECOOP 2010: Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *LNCS*, pages 405–428. Springer, 2010.

[29] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[30] Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *SIGGRAPH Comput. Graph.*, 19:199–204, July 1985.

[31] Cédric Champeau. Groovy enhancement proposal 10 - static compilation. Available online: `http://docs.codehaus.org/display/GroovyJSR/GEP+10+-+Static+compilation` (accessed November 15th 2012), 2012.

[32] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[33] Adrian Colyer. AspectJ. In *Aspect-Oriented Software Development*, pages 123–143. Addison-Wesley, 2005.

[34] Oracle Corporation. JSR 335: Lambda expressions for the Java programming language, version 0.5.1. Technical report, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, U.S.A., 2012.

[35] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical Report Publication No. S-22 (Revised edition of publication S-2), Norwegian Computing Center, October 1970.

[36] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, September 1966.

[37] Peter J. Denning. Is computer science science? *Commun. ACM*, 48(4):27–31, April 2005.

[38] Edsger Wybe Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[39] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 51–62, New York, NY, USA, 2009. ACM.

[40] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, April 2002.

[41] ECaesarJ homepage. Available online: `https://gforge.inria.fr/projects/ecaesarj/` (accessed October 23rd 2012).

[42] Robert Eckstein. Mixins in JavaFX 1.2 technology. Sun Microsystems. Available online: `http://www.oracle.com/technetwork/articles/javafx/mixin-137619.html` (accessed October 23rd 2012), 2009.

[43] Ecma International. *Standard ECMA-334 C# Language Specification*, 4th edition, 2006.

[44] Ecma International. *Standard ECMA-364 Eiffel: Analysis, Design and Programming Language*, 2nd edition, June 2006.

[45] Erik Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999. Phd Thesis, Department of Computer Science, University of Aarhus, Denmark.

[46] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[47] Erik Ernst. The expression problem, Scandinavian style. In Philippe Lahire, Gabriela Arévalo, Hernán Astudillo, Andrew P. Black, Erik Ernst, Marianne Huchard, Markku Sakkinen, and Petko Valtchev, editors, *MASPEGHI 2004*, 2004.

[48] Ada Europe. *Ada Reference Manual*. 2006. ISO/IEC 8652:1995(E).

[49] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[50] Robert Field. JavaFX language reference. Sun Microsystems. Available online: `http://openjfx.java.sun.com/current-build/doc/reference/JavaFXReference.html` (accessed October 23rd 2012).

[51] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Rober E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–31. Addison-Wesley, 2005.

[52] Django Software Foundation. The Django framework version 1.4.3. Available online: `https://www.djangoproject.com/download/1.4.3/tarball/` (accessed October 23rd 2012).

[53] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[54] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17:145–205, March 2007.

[55] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel N. Nez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[56] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into Java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 73–90, New York, NY, USA, 2008. ACM.

[57] Brian Goetz. Interface evolution via virtual extension methods, third draft. `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v3.pdf` (accessed October 23rd 2012), 2011.

[58] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Specification - Java SE 7 Edition*. Oracle America Inc, 500 Oracle Parkway M/S 5op7, California 94065, USA, 2012.

[59] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proc. Object-oriented programming, systems, languages, and applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[60] Roy Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2010.

[61] Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl. A semantics-based aspect language for interactions with the arbitrary events symbol. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 262–277, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl. Semantics-based weaving of UML sequence diagrams. In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 122–136, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] Groovy: Runtime vs compile time, static vs dynamic. Available online: `http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic` (accessed November 15th 2012).

[64] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18:109–138, March 1996.

[65] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.

[66] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[67] Øystein Haugen, Stein Krogdahl, and Birger Møller-Pedersen. SWAT: Semantics-preserving Weaving - Advancing the Technology. Research Project Proposal. Department of Informatics, University of Oslo. 2005.

[68] Øystein Haugen and Ketil Stølen. Stairs - steps to analyze interactions with refinement semantics. In *of Lecture Notes in Computer Science*, pages 388–402. Springer, 2003.

[69] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 36–56, London, UK, UK, 1993. Springer-Verlag.

[70] Einar Høst and Bjarte Østvold. Debugging method names. In Sophia Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317. Springer Berlin / Heidelberg, 2009.

[71] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. *SIGPLAN Not.*, 43(6):79–89, June 2008.

[72] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 185–198, New York, NY, USA, 2007. ACM.

[73] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[74] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *LNCS*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[75] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[76] Stein Krogdahl. Generic packages and expandable classes. Research Report 298, Department of Informatics, University of Oslo, October 2001.

[77] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[78] Guillaume Laforge. What's new in groovy 2.0? Published on InfoQ Jun 28, 2012. Available online: `http://www.infoq.com/articles/new-groovy-20` (accessed November 15th 2012), 2012.

[79] Guillaume Laforge and The Gaelyk development team. Gaelyk toolkit. Available online: `http://gaelyk.appspot.com/` (accessed October 23rd 2012).

[80] Greg R. Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[81] Henry Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[82] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole. *Java Swing*. O'Reilly Media, 2012.

[83] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[84] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, New York, NY, USA, 1993.

[85] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[86] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 52–67, New York, NY, USA, 2002. ACM.

[87] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proc. Aspect-Oriented Software Development '03*, pages 90–99, New York, 2003. ACM.

[88] Mira Mezini and Klaus Ostermann. Untangling crosscutting models with CAESAR. In Rober E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Proc. Aspect-Oriented Software Development '05*, pages 165–199. Addison-Wesley, 2005.

[89] Microsoft Corporation. *C# Language Specification Version 4.0*, 2010.

[90] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 279–303, London, UK, UK, 1999. Springer-Verlag.

[91] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[92] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[93] Angel Nunez and Vaidas Gasiunas. ECaesarJ user's guide, September 2009.

[94] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[95] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[96] Martin Odersky. The Scala language spec. version 2.9 – draft, 2011.

[97] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the Scala programming language. Technical report, 2004.

[98] Jon Oldevik. *Semantics Preservation in Model-based Composition*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2010.

[99] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proc. Object-oriented programming, systems, languages, and applications*, OOPSLA 2010, pages 341–360, New York, NY, USA, 2010. ACM.

[100] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. *SIGPLAN Not.*, 30(10):235–250, 1995.

[101] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, January 2008.

[102] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. European Conference on Object-Oriented Programming*, pages 419–443, 1997.

[103] The Python Software Foundation. *The Python Language Reference*, 2.7.3, version dec 11, 2012 edition, 2012.

[104] Ruby on rails version 3.2. Available online: `https://github.com/rails/rails/tree/3-2-stable` (accessed October 23rd 2012).

[105] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, pages 157–188, 2005.

[106] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[107] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, December 2001.

[108] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423 – 465, 2008. Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).

[109] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, 21(11):38–45, June 1986.

[110] Ida Solheim and Ketil Stølen. Technology research explained. Research Report A313, SINTEF ICT, 2007. Available online: `http://heim.ifi.uio.no/ketils/kst/technical-reports.htm` (accessed October 23rd 2012).

[111] Fredrik Sørensen, Eyvind W. Axelsen, and Stein Krogdahl. Dynamic composition with package templates. In *Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, volume 564. CEUR-WS.org, March 2010.

[112] Fredrik Sørensen, Eyvind W. Axelsen, and Stein Krogdahl. Reuse and combination with package templates. In *The MASPEGHI Workshop at ECOOP 2010 – MechAnisms for SPEcialization, Generalization and inHerItance*, June 2010.

[113] J. Soukup. Implementing patterns. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, Boston, 1995.

[114] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[115] Springsource. Grails framework version 2.1.2. Available online: `http://dist.springframework.org.s3.amazonaws.com/release/GRAILS/grails-2.1.2.zip` (accessed October 23rd 2012). Springsource is a division of VMware.

[116] Springsource. Groovy version 1.8.8. Available online: `http://dist.groovy.codehaus.org/distributions/groovy-src-1.8.8.zip` (accessed October 23rd 2012). Springsource is a division of VMware.

[117] SHARPCRAFTERS s.r.o. PostSharp version 2.1. Available online: `http://www.sharpcrafters.com/downloads/postsharp-2.1` (accessed October 23rd 2012).

[118] Håkon Stordahl. BooPT: Implementasjon av package templates for Boo. Master's thesis, Deptartment of Informatics, University of Oslo, Gaustadalléen 23B, Postboks 1080 Blindern, 0316 Oslo, 2012.

[119] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, Berlin, Germany, September 2004.

[120] Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, 1989.

[121] Bjarne Stroustrup. The C++0x "remove concepts" decision. *Dr. Dobb's*, July 2009. Available online: `http://www.drdobbs.com/architecture-and-design/218600111` (accessed October 23rd 2012).

[122] Bjarne Stroustrup. Simplifying the use of concepts. *JTC1/SC22/WG21 - The C++ Standards Committee*, 2009. Available online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf` (accessed October 23rd 2012).

[123] Fredrik Sørensen and Stein Krogdahl. Generic packages with expandable classes compared with similar approaches. In Jon Markus Bjørndalen and Frode Eika Sandnes, editors, *NIK 2007*. Tapir akademisk forlag, 2007.

[124] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 168–179, New York, NY, USA, 2008. ACM.

[125] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM.

[126] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 27–46, New York, NY, USA, 2003. ACM.

[127] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[128] Dave Thomas and Andy Hunt. *Programming Ruby - The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc, 2001.

[129] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, ECOOP '97, pages 444–471. Springer-Verlag, 1997.

[130] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 186–204, London, UK, 1999. Springer-Verlag.

[131] Mads Torgersen. Virtual types are statically safe. In *In Proceedings of FOOL '98: 5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[132] Mads Torgersen. The expression problem revisited. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143. Springer, 2004.

[133] Jan Vitek. The rise of dynamic languages. Slides from summer school talk at ECOOP 2011. Available online: `http://scc-sentinel.lancs.ac.uk/ecoop11/?q=content/rise-dynamic-languages` (accessed October 23rd 2012), 2011.

[134] Phillip Wadler. The expression problem, 1998. Posted on the Java Genericity mailing list. Available online: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt` (accessed October 23rd 2012).

[135] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 37–56, New York, NY, USA, 2006. ACM.

[136] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized Interfaces for Java. In *European Conference on Object-Oriented Programming (ECOOP) 2007, Proceedings*, LNCS. Springer-Verlag, July 2007.

[137] Stefan Wehr and Peter Thiemann. JavaGI in the battlefield: practical experience with generalized interfaces. In *Proc. eighth international conference on Generative programming and component engineering*, GPCE '09, pages 65–74, New York, NY, USA, 2009. ACM.

[138] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973.

Part II

Research Papers

Chapter 8

Paper I: A Reusable Observer Pattern Implementation Using Package Templates

Authors. Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl.

Publication. Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2009.

Abstract. In this paper, we show how *package templates*, a new mechanism for code modularization, paired with a comparatively (and intentionally) small AOP mechanism may be utilized to create a reusable package for the Observer design pattern that can be plugged into an existing architecture with a minimum of "glue code".

Categories and Subject Descriptors. D.3 **[Software]**: Programming Languages; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Classes and objects*; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Patterns*

General Terms. Languages, Design.

Keywords. Aspects, AOP, OOP, Templates, Design Patterns.

8.1 Introduction

The concept of design patterns [6] is an approach to object oriented design and development that attempts to facilitate the reuse of conceptual solutions for common functionality required by certain *patterns* or classes of common problems. As such, they seem like a perfect candidate for inclusion as reusable components in frameworks, infrastructure software, etc. However, it seems that even though the concepts of many patterns are in relative widespread use, implementing them as reusable components in mainstream languages like C# or Java is hard. This is at least in part due to limitations with regard to e.g. *extensibility of the subtyping relation* [13] and/or lack of support for mechanisms dealing with crosscutting concerns.

One commonly used design pattern is the Observer pattern. A few implementations utilizing various new language extensions already exist, notably one by Hannemann and Kiczales [8] utilizing AspectJ [2], and one by Mezini and Ostermann [11] utilizing the Caesar system [1].

The package template (PT) mechanism [9, 10, 17] targets the development of collections of reusable interdependent classes. Such a collection is a template for a package, that may be *instantiated* at compile time, thus forming an ordinary package. At instantiation, the template may be customized according to its usage, and classes from different independent templates may be merged to form one new class.

In this article, we utilize package templates with a comparatively (and intentionally) small and simple aspect oriented extension to provide a reusable package for the Observer pattern, and compare our approach to the other solutions mentioned above.

8.2 Overview of the PT Mechanism

We here give a brief and general overview of the package template mechanism. The concepts of the mechanism are not in themselves tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax.

A package template looks like a regular Java package, but we will use a syntax where curly braces enclose the contents of both templates and regular packages., e.g.:

```
template T<E> {
  class A { ... }
  class B extends A { ... }
}
```

Templates may have (constrained) type parameters, such as `E` above, but this will not be treated in any detail in this paper. Apart from this and a few other constructs (such as the extensions we propose in Section 8.2.1), valid contents of a template are also valid as plain Java programs. As such, templates may also be type checked independently of their potential usage(s).

In PT, a template is instantiated at compile time with an `inst` statement, which has some significant differences from Java's `import`. Most notably, an instantiation will

create a local copy of the template classes, potentially with specified modifications, within the instantiating package. An example of this is shown below:

```
package U {
  inst T<C> with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D extends C since B extends A
}
```

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `U`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`. The example above additionally shows how the template classes `A` and `B` are renamed to `C` and `D`, respectively, and that expansions are made to these classes. Expansions are written in `adds`-clauses, and may add variables and methods, and also override virtual or implement abstract methods from the template class.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is *re-typed* to the corresponding expansion classes (`C` and `D`) at the time of instantiation (PT rules guarantee that this is type-safe). Any sub/super-type relations within the template is preserved in the package where it is instantiated.

Another important property is that classes from different, possibly unrelated, templates may also be *merged* upon instantiation to form one new class. Consider the simple example below:

```
template T {
  class A { int i; A m1(A a) { ... } }
}

template U {
  abstract class B { int j; abstract B m2(B b); }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;

class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the abstract `m2` from `B` is implemented

in the `adds` clause, and furthermore how both `m1` and `m2` now have signatures of the form `MergeAB` → `MergeAB`. We shall use a similar construct in Section 8.3.

To sum up, some of the useful properties of PT are: It supports writing reusable templates of interdependent, cooperating classes which may be statically type checked without any information of their usage. Upon instantiation, a template class may be customized, and merged with other template classes. References within a template to a template class will be re-typed according to the instantiation.

8.2.1 AOP Extensions

We here extend the basic PT mechanism described above with a minimal set of constructs for aspect-oriented programming. Thus, we consider a restricted version of common AOP concepts, that paired with the inherent possibilities in PT for e.g. merging and expanding classes may provide some of the power and flexibility found in "traditional" AOP languages.

"Aspect-oriented programming is quantification and obliviousness" [5] is often seen as an imperative quote for what AOP is or should be. However, in this paper we wish to investigate the possible benefits of a slightly different approach. To begin with, aspects are realized not as separate entities (neither at runtime nor compile-time), but rather as pointcuts and advice defined as members of (template) classes. In practice, this means that the possible changes to a base program by a (conceptual) aspect will be limited in scope by a corresponding template instantiation. Furthermore, it means that the pointcuts are local to the defining class, including its subclasses (given the appropriate modifier), and that they may only refer to members within the defining class or any of its superclasses, following the OO principle of encapsulation. Hence, they may also be refined or redefined by subclasses or in addition clauses. The same will apply to advice.

This may seem like a severe restriction compared to e.g. AspectJ, but we believe that paired with the flexible tailoring mechanisms offered by PT (in particular the merging possibilities), this provides a sufficiently powerful and expressive construct for many purposes.

Given that all pointcuts will refer to local members, *quantification* using wild-cards over member or type names should not be as necessary as in e.g. AspectJ. Therefore, we find it worthwhile to explore the disallowing of wild-card usage with regard to member names, and rely instead on explicit join point specification. This will result in a mechanism where the pointcuts do not specify a pattern to be matched against join points, but instead an actual *binding* to join points.

Pointcuts are declared inside classes according to the following EBNF grammar sketch, where terms in quotes are terminals and the unquoted ones are non-terminals. Productions that are equal to their Java equivalents are left out for brevity, and things enclosed in << and >> should be understood as "pseudo-EBNF" in place of parts left out:

```
pc_decl ::= { modifier } "pointcut"
 ( qualified_identifier | "void" | "*" )
 identifier "(" [ <<parameter list>> ] ")"
 ( "{" pc_expr "}" | ";" )

pc_expr ::=
 ( "call" | "execution" | "get" | "set" ) "(" identifier ")"
 | pc_expr "&&" pc_expr
 | pc_expr "||" pc_expr | "(" pc_expr ")"
```

Likewise, an advise has the following syntax:

```
advice_declaration ::=
  { modifier } "advice" identifier
  ( "before" | "after" | "around" ) identifier
  "{" { <<valid statement>> } "}"
```

To keep this exposition short, we will not discuss the EBNF or the generated language in any further detail, but rather turn to a small example to illustrate how the mechanism works. Consider the following template:

```
template T {
  class A {
    A m1(A a) { ... }
    pointcut A pc1 (..) { call(m1) }
}}
```

The class `A` contains a pointcut that matches calls to the the method `m1`. The template may be instantiated as shown below:

```
inst T with A => B;
class B adds {
  pointcut B pc1(..) { call(m1) || call(m2) }
  advice afterM after pc1 { ... }
  B m2() { ... }
}
```

Here, `A` is re-typed to `B`, and `B` adds a method and an advice, and refines the pointcut `pc1`. Note that even though `pc1` in `T` refers to the type `A`, the pointcut will after instantiation refer to `B`, and continue to match method calls to `m1`, and this would hold even if `m1` was renamed in the instantiation. This is made possible by the fact that pointcuts are not string patterns, but actual bindings, as mentioned above.

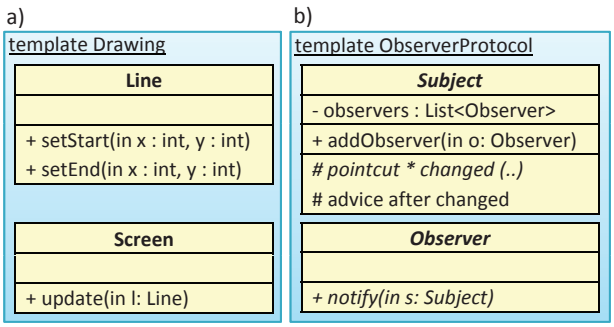

Figure 8.1: a) The Drawing template, and b) the roles of the Observer pattern

8.3 The Observer Pattern Example

The observer pattern is a design pattern with two roles, the subject and the observer. Each subject maintains a list of observers that are interested in being notified when certain (yet unspecified) changes occur in the subject. An observer may choose to observe one or more subjects at any given time.

8.3.1 Single Subject/Single Observer Classes

To exemplify the use of the pattern, we consider a package for drawing objects on a screen (strongly resemblant of the examples in [8] and [11]). In this section, we look at an example with only two classes, `Screen` and `Line`, as shown in Figure 8.1a. When a line changes its length or position, the screen should be updated to reflect the changes. We would like this logic to be abstracted out of the concrete `Screen` and `Line` classes, so that it can be reused for other manifestations of this particular problem.

Utilizing package templates, we could implement the Observer pattern as shown in Figure 8.1b with the following code[1]:

```
template ObserverProtocol {
 public abstract class Observer {
   abstract void notify(Subject changee);
 }
 public abstract class Subject {
  List<Observer> observers = new List<Observer>();

  public void addObserver(Observer o) {observers.add(o);}
```

[1]The removeObserver method is trivial and hence omitted for brevity. The classes in the diagrams contain a fourth compartment for AOP related members where this is relevant, and the names of abstract classes and members are written in italics.

```
  abstract protected pointcut * changed(..);
  protected advice ac after changed {
    foreach(Observer o in observers) { o.notify(this); }
} } }
```

The `Observer` class has only one method, the abstract `notify`, that will have to be implemented at a later stage to make it meaningful to the concrete observers.

The subject class has methods for adding and removing observers from the current subject instance. Furthermore, it defines an abstract pointcut `changed`, that will have to be refined in concrete subjects. The pointcut specifies that any parameters and return types are valid for its (as of yet undefined) corresponding join points with the use of wild-cards "`*`" and "`..`". Using the drawing package of Figure 8.1a, we could do the instantiation as follows:

```
inst Drawing with
  Screen => ScreenObserver, Line => LineSubject;
inst ObserverProtocol with
  Observer => ScreenObserver, Subject => LineSubject;
```

`ScreenObserver` and `LineSubject` become a merge of `Observer` and `Screen`, and `Subject` and `Line`, respectively. Furthermore, we make use of `adds` clauses to concretize the abstract members of the template classes. These clauses are only needed in order to concretize what was left as abstract in the respective templates.

```
class ScreenObserver adds {
 void notify(LineSubject l) { update(l); }
}
class LineSubject adds {
 pointcut * changed(..) {call(setStart) || call(setEnd)}
}
```

The two resulting classes are shown graphically in Figure 8.2. Note that that a retyping has occurred, such that for instance the `addObserver` method now takes a `ScreenObserver` as its only parameter. Similarly, the `notify` method now takes a `LineSubject` parameter. Due to the retyping done by the PT mechanism, no casts are required.

8.3.2 Multiple Subject and/or Observer Classes

In the previous section we looked at a rather simple scenario in which there was only one class having the subject role, and one having the observer role. In this section, we will look at the more general problem, with multiple classes playing the roles of subjects and observers.

To exemplify, we extend the template `Drawing` such that there are several classes acting, respectively, as subjects and observers, and hence we need to modify our instantiation code a little. The good news, however, is that our implementation of the

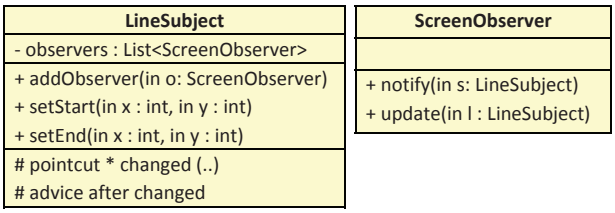

Figure 8.2: The resulting classes from the merge of the ObserverProtocol and Drawing templates.

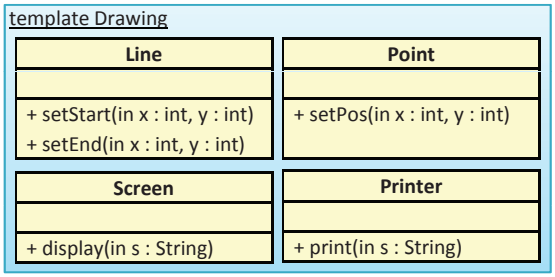

Figure 8.3: The drawing template with two potential subjects and two potential observers

Observer pattern itself needs no modification. The classes of the new `Drawing` template are shown in Figure 8.3. Applying our Observer pattern to this package, we would use the following instantiation:

```
inst Drawing with
  Screen => ScreenObserver,
  Printer => PrinterObserver,
  Line   => LineSubject,
  Point   => PointSubject;

inst ObserverProtocol with
  Observer => BaseObserver,
  Subject  => Figure;
```

The `Drawing` template does not contain any base classes into which the required functionality for subjects and observers can be merged. However, PT allows us to *introduce* superclasses directly, in a type safe manner, as shown in the instantiations above and the addition classes below:

```
abstract class BaseObserver adds {}

class ScreenObserver extends BaseObserver adds {
 void notify(Figure f) { display(f + " has changed"); }
}

class PrinterObserver extends BaseObserver adds {
 void notify(Figure f) { print(f + " has changed"); }
}

abstract class Figure adds {}

class LineSubject extends Figure adds {
 pointcut * changed(..) {call(setStart) || call(setEnd)}
}

class PointSubject extends Figure adds {
 pointcut * changed(..) { call(setPos) }
}
```

The resulting package is shown in Figure 8.4. Note how the functionality for the Subject and Observer roles are distributed to their respective functional counterparts from the `Drawing` package through the newly introduced common superclasses.

8.4 Related Work

The original description of the Observer pattern by the so-called "Gang of Four" (GoF) found in [6] comes with an example implementation in C++. This implementation makes use of C++'s support for multiple inheritance to provide subject and observer classes for the pattern. Since we are targeting Java in this example, such inheritance hierarchies cannot be used.

The GoF version also makes use of a special "trick" in which the observer knows the identity of the (single) subject for changes in which it is interested. Because of this they can avoid having to do a type cast that would have been necessary in the general case.

In *Design Pattern Implementation in Java and AspectJ* [8], Hannemann and Kiczales implement the design patters from the GoF in AspectJ, and contrast these implementations to their potential pure Java counterparts. The Observer pattern is used as a running example. In their implementation, the pattern is handled by one single abstract aspect, the `ObserverProtocol`. This aspect will exist as an entity at runtime, and contains a global map of observers to subjects. This is in contrast to the PT version, in which there is no notion of the aspect as a separate entity at runtime, but rather that the roles of the aspect are imposed on (and localized in) the classes that are to play the respective roles.

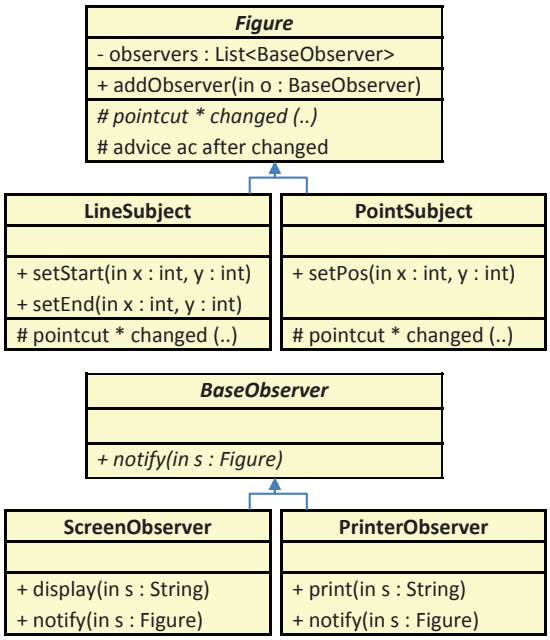

Figure 8.4: The resulting classes from the merge of the ObserverProtocol template and the new version of the Drawing template

The AspectJ aspect defines empty "marker" interfaces, that conceptually declare the existence of the two roles of the pattern. However, since the interfaces are empty, the aspect does not make it explicit which operations/behaviors belong to either participant in the pattern.

The abstract aspect is concretized via inheritance as concrete aspects, that override the abstract pointcut `subjectChange`, and define which concrete classes should be designated as either subject or observer through the AspectJ `declare parents` construct. Different concrete aspects may be defined based on the abstract aspect. In order to update the observer(s) when a change has occurred, a runtime cast must be made from the interface `Observer` to the class `Screen`, since the interface is empty and hence has no knowledge of the `Screen`'s `display` method.

In PT, the concretization happens through the `inst` statement (with optional addition classes for e.g. concretizing abstract members). The retyping may in many cases eliminate the need for casts completely.

The fact that pointcuts are not explicitly bound to join points, means that the AspectJ version is more prone to errors stemming from faulty pointcuts (the *fragile pointcut problem* [16]), while the PT version sacrifices some flexibility for a greater degree of relative pointcut safety.

AspectC++ [15] adds full-blown AOP support to the C++ language, including support for templates and non-OO programming. Contrary to our approach, the authors explicitly state as one of their goals to *"not make any compromise regarding obliviousness and quantification"*, i.e. to not put any restrictions on their language with regard to the definition from [5]. This provides for a very powerful paradigm, which pairs up well with the nature of C++ itself.

One of the examples in this article shows a reusable Observer pattern implementation. It makes use of C++'s multiple inheritance capabilities and a distinguishing join point API to add the required members to the classes designated as subject and observer, respectively. The solution is similar to ours in the respect that only pointcuts and the update method for the observer need to be concretized for a particular utilization, yet differ in the sense that while our implementation syntactically distributes these members to their respective classes, the AspectC++ version keeps everything in a central aspect.

In *Conquering Aspects With Caesar* [11], Mezini and Ostermann show an alternative implementation of the pattern, utilizing the CaesarJ system. The work mainly addresses two points with regard to the AspectJ implementation discussed above; (I) the need for expressing aspects not as a monolithic entity, but rather as a set of interrelated, interacting modules, and (II) the need for flexible and reusable aspect bindings and implementations.

With respect to (I), Caesar achieves this through so-called *aspect collaboration interfaces* (ACIs). The ACIs are hierarchical, and may contain several (related) subinterfaces. Each interface may describe a set of provided and/or required operations. The provided operations must be realized by implementers of the interface (e.g. the `addObserver` method must be implemented by an implementation of the `Subject` interface). This brings us over to point (II).

The required operations of an ACI must be implemented by a *binding* (which is separate from the implementation discussed in the previous paragraph), that binds the ACI to classes in the base program. Bindings may be defined independently of ACIs and their implementations, offering greater flexibility than the AspectJ counterpart.

In the PT example, the template is the rough equivalent of both an ACI and its implementation. We could, however, have used interfaces for subjects and observers to separate actual implementation from public interface, but we are not obliged to.

A CaesarJ binding can be compared to the `inst` statement (with optional addition classes) in PT, in the sense that required/abstract members will be concretized, and roles will be mapped to base program classes.

Like in Caesar (and in contrast to AspectJ), aspects in PT must be *explicitly deployed*. That is, the mere inclusion of an ACI or a package template that contains pointcut and advice in the compilation process does not alter any part of the base program. For this to happen in PT, an explicit `inst` statement must be present. Similarly, Caesar employs the `deploy` keyword for much the same purpose, with a couple of important distinctions: Caesar supports aspectual polymorphism and dynamic as well as static deployment of aspects, while we (at present) only support static instantiation of templates. However, since every instantiated template class will correspond to an actual class in the base program, and pointcuts and advice reside within ordinary classes, ordinary OO polymorphism may possibly be used in such situations, though we have not yet explored this issue in any detail.

Concerning the PT mechanism itself, several others have defined mechanisms that in some ways are similar (disregarding the AOP extensions presented here). Examples are J& [12], Classboxes [3], Traits [14] and Mixins [4]. For a more thorough treatment of their respective similarities and differences compared to PT, the interested reader is referred to [17].

8.5 Conclusion and Future Work

We have suggested how package templates extended with a (compared to its contemporaries) rather minimal AOP mechanism may provide a sufficiently powerful framework for implementing reusable components that map to interdependent entities in a base program. Exemplified here through the Observer design pattern, this is clearly an area that we hope to generalize and expand on.

One interesting topic in that respect is to investigate further the interplay between PT and AOP through experimenting with different degrees of AOP complexity with regard to e.g. pointcut scope and expressiveness, and see how different points along this axis pair up with inherent PT mechanisms such as re-typing and merging.

Furthermore, it would be interesting to try to assess how applicable and useful the constructs we have used here really are when applied to a broader set of real-world problems. With that in mind, it would be interesting to relate the findings from [7] to an implementation of the same patterns in PT.

Acknowledgements

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30). We would like to thank the anonymous reviewers for valuable comments.

Bibliography

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] AspectJ Team. The AspectJ programming guide, 2003.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05*, pages 177–189, New York, 2005. ACM.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[5] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Rober E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *AOSD*, pages 21–31. Addison-Wesley, 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05*, pages 3–14, New York, 2005. ACM.

[8] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173, 2002.

[9] Stein Krogdahl. Generic packages and expandable classes. Technical Report 298, Department of Informatics, University of Oslo, 2001.

[10] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. To appear, 2009.

[11] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03*, pages 90–99, New York, 2003. ACM.

[12] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, 2006. ACM.

[13] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121 – 145, 2008.

[14] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[15] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[16] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, Berlin, Germany, September 2004.

[17] Fredrik Sørensen and Stein Krogdahl. Generic packages with expandable classes compared with similar approaches. In *NIK 2007*. Tapir akademisk forlag, 2007.

Chapter 9

Paper II: Towards Pluggable Design Patterns Utilizing Package Templates

Authors. Eyvind W. Axelsen and Stein Krogdahl.

Publication. Proceedings of Norsk Informatikkonferanse (NIK) 2009.

Abstract. In this paper, we show how *package templates*, a new mechanism for code modularization, may be utilized to create reusable packages for selected design patterns that can be plugged into an existing architecture with a minimum of "glue code". Exemplified through these implementations we consider some desirable extensions to the previously published version of the mechanism.

9.1 Introduction

The concept of design patterns [9] is an approach to object oriented design and development that attempts to facilitate the reuse of conceptual solutions for common functionality required by certain *patterns* or classes of common problems. As such, they seem like a perfect candidate for inclusion as reusable components in frameworks, be they completely general such as e.g. the .NET framework [17] or application specific. However, it seems that even though the concepts of many patterns are in relative widespread use, implementing them as reusable components in mainstream languages like C# [6] or Java [11] is hard. This is at least in part due to limitations with regard to e.g. *extensibility of the subtyping relation* [19] and/or lack of support for mechanisms dealing with crosscutting concerns.

A few existing papers explicitly discuss new language concepts in the context of implementing design patterns, notably one by Hannemann and Kiczales [10] utilizing AspectJ [2, 5], and one by Mezini and Ostermann [16] utilizing the Caesar system [1].

The package template (PT) mechanism [12, 13] targets the development of collections of reusable interdependent classes. Such a collection is a template for a package, that may be *instantiated* at compile time, thus forming an ordinary package. At instantiation, the template may be customized according to its usage, and classes from different independent template instantiations (of a single or several distinct templates) may be merged to form one new class.

In this article, we utilize package templates to provide reusable implementations for a few selected design patterns. Exemplified through these pattern implementations, we will consider a few desirable extensions to PT.

9.2 Overview of the Package Template Mechanism

We here give a brief and general overview of the package template mechanism. The concepts of the mechanism are not in themselves tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax, and the forthcoming examples will be based on Java.

A package template looks like a regular Java file, but we use a syntax where curly braces enclose the contents of templates, e.g. as follows:

```
template T<E> {
  class A { ... }
  class B extends A { ... }
}
```

Templates may have (constrained) type parameters, such as `E` above, but we will not make use of this feature in this paper. Apart from this and a few other constructs (including the extensions we propose), valid contents of a template are also valid as plain Java programs. As such, templates may also be type checked independently of their potential usage(s).

In PT, a template is instantiated at compile time with an `inst` statement, which has some significant differences from Java's `import`. Most notably, an instantiation will create a local copy of the template classes, potentially with specified modifications, within the instantiating package. An example of this is shown below:

```
package P;
inst T<C> with A => C, B => D;
class C adds { ... }
class D adds { ... } // D extends C since B extends A
```

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `P`.[1] In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`, without any other clauses. The example above additionally shows how the template classes `A` and `B` are renamed to `C` and `D`, respectively, and that expansions are made to these classes. Expansions are written in `adds`-clauses, and may add variables and methods, and also override virtual or implement abstract methods from the template class.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is *re-typed* to the corresponding expansions (`C` and `D`) at the time of instantiation (PT rules guarantee that this is type-safe). Any sub/super-type relations within the template is preserved in the package where it is instantiated.

Another important property is that classes from different, possibly unrelated, templates may also be *merged* upon instantiation to form one new class. Consider the simple example below:

```
template T {
  class A { int i; A m1(A a) { ... } }
}
template U {
  abstract class B { int j; abstract B m2(B b); }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;

class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

[1] In the following, we will skip the explicit package declaration in the examples.

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the abstract `m2` from `B` is implemented in the `adds` clause (and the expansion does hence not need to be declared abstract), and furthermore that both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`.

To sum up, some of the useful properties of PT are: It supports writing reusable templates of interdependent, cooperating classes which may be statically type checked without any information of their usage. Upon instantiation, a template class may be customized, and merged with other template classes. References within a template to a template class will be re-typed according to the instantiation.

9.3 Design Pattern Implementation and Extensions to PT

In this section, we will look at how three design patterns may be implemented in PT with a few extensions that were deliberately not treated in [13]. Though they are exemplified here through pattern implementations, it is our belief that the extensions are generally usable and viable as integral parts of PT.

9.3.1 Running Example

To exemplify usage of the design patterns we consider below, we will use a simple example that revolves around two classes, `Screen` and `Line`. The idea is that a screen may be used to draw and display lines. This may be (naïvely) implemented by the following package template sketch:

```
template Drawing {
  public class Screen {
    public void update(Line l) { ... }
    ...
  }
  public class Line {
    protected int x1, y1, x2, y2;
    public void setStart(int x, int y) { x1 = x; y1 = y; }
    public void setEnd(int x, int y) { x2 = x; y2 = y; }
    ...
} }
```

9.3.2 The Singleton Pattern

The Singleton pattern is a pattern from [9] that is used to restrict the number of objects of a class to at most one at any given time. This is typically done by making the constructor of the class private, and providing a static `Instance` property, through which the object may be accessed. The object is typically created lazily when requested for the first time.

Despite the inherent conceptual simplicity of this pattern, there are several pitfalls to be aware of when implementing it, that, depending on the language used, might make it tricky to get right (see for instance [20] and [21] for discussions pertaining to the Java memory model, thread safety and more). This points to the need for a safe and reusable implementation. Generic constructs help, but are (in current implementations of Java) unable to express the constraint that the singleton should have a private constructor. With a package template, this can be expressed quite naturally, e.g.:[2]

```
template SingletonPattern {
   public class Singleton {
      private Singleton() {}

      // SingletonHolder is loaded on the first execution of
      // getInstance() or the first access to
      // SingletonHolder.INSTANCE, not before.
      private static class SingletonHolder {
        private static final Singleton INSTANCE =
          new Singleton();
      }

      public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
} } }
```

Visibility and access modifiers for template classes is a topic that, save for a few suggestions, is explicitly left for future work in [13]. While we do not have room for a full treatment of this here, we follow and expand on these suggestions, and shall use the following rules:

With respect to access modifiers, a template is considered a separate package. That is, the default Java accessibility ('package-private') means within the template itself. Protected members are visible to subclasses, addition clauses and merged classes. Private members are visible only to the declaring class, with one exception: private constructors are visible to addition clauses, and may be called from constructors declared there. This is vital, since it is required in PT to include a new constructor for merged classes that differ in this respect. However, the addition class may not create new objects using the `new` operator under such conditions, and it may also not increase the visibility of the constructor (e.g. the addition of a public constructor would not be allowed in the example below). Thus, if a template class `TC` with a private constructor is given an addition clause `D` in an instantiation, then objects of type `D` can only be generated by a statement `new TC(...)` from within the template.

Now, to reuse the pattern implementation from above and make a domain class into a singleton, we may simply instantiate the template and map the `Singleton` class to a fitting domain abstraction, e.g. a factory for `Screen`s from our running example.

[2]Example implementation of the Singleton class adapted from the Wikipedia implementation available at `http://en.wikipedia.org/wiki/Singleton_pattern`, accessed June 22, 2009.

```
inst SingletonPattern with Singleton => ScreenFactory;
inst Drawing;

class ScreenFactory adds {
  public Screen createScreen() {
    Screen s = ... < create new or reuse existing screen > ...
    return s;
} }
class Program {
   static void main(String[] args) {
      Screen s = ScreenFactory.getInstance().createScreen();
      s.update(new Line());
      ...
} }
```

The `ScreenFactory` class will now have all the properties of a singleton according to the pattern (including a private constructor) as well as the added `createScreen` method.

This example shows that, through the use of package templates, we may utilize the full capability of the Java (or other targeted) language to specify both constraints and functionality for a generic (in a broad sense of the word) reusable implementation, which goes beyond what is available in the language today. Using techniques similar to the one described above, similar implementations of e.g. the *Flyweight* and *Multiton* patterns from [9] could also be made.

9.3.3 Nested Classes and the Memento Pattern

The Memento pattern is a design pattern from [9] that revolves around three roles; the *originator*, the *caretaker* and the *memento*. The caretaker is, at different points in an application, interested in storing and managing the *state* of the originator. However, an important point of this pattern is that the state of the originator should not be exposed to caretaker. Hence, the memento provides a way to allow the caretaker to store the state of the originator, without knowing its inner details. The pattern could e.g. typically be used to provide an application with undo/redo functionality, and hence the originator provides methods to both retrieve and restore its state by using mementos.

This pattern may be implemented as a reusable 'skeleton' in PT, as shown below. While this template leaves the bulk of the actual implementation up to users of the template, it provides the benefit of clearly laying out the pattern's participants, and providing well defined (abstract) points in the code that should be concretized by the user.

```
template MementoPattern {
  public abstract class Originator {
    public Memento CreateMemento() {
       return new Memento(GetState());
    }

    public abstract void SetMemento(Memento m);
    protected abstract State GetState();
    protected abstract class State { }

    public class Memento {
      private Memento(State state) { this.state = state; }
      private State state;
} } }
```

This implementation does not explicitly involve the caretaker role, since objects having this role would merely be consumers of the public interface provided by the template. A skeleton caretaker implementation that made use of this interface could be provided if desirable.

The `Originator` class contains nested classes `State` and `Memento`, where the former is abstract. They conceptually belong to the originator, so nesting makes good sense from a code organizational point of view.

Template classes that contain nested classes is an issue that is not treated in [13]. However, since this is an integral part of Java, that appears to be in widespread use both for code organization in general and, in particular, for event handling, it seems like a natural extension of the mechanism, that may additionally provide some of the benefits of *virtual classes* [14], due to the possibilities provided by addition clauses. When implementing support for this in Java PT, care must be taken so that e.g. references to `<EnclosingClass>.this` are re-typed correctly. Note that this also applies to anonymous inner classes, even though these are not allowed to have addition clauses or be merged with other classes. Furthermore, when merging an inner class with another class, care must be taken (by the PT framework) to not introduce any inconsistencies in the inheritance hierarchy.

Consider the following example template containing nested classes:

```
template Nested {
   class Outer{ class Inner { ... } ... }
}
```

Templates such as this may be instantiated in the normal manner, with possibilities for addition clauses or merges of both `Outer` and `Inner`, e.g. as follows:

```
inst Nested with Outer => A { Inner => B }
class A adds { ... class B adds { ... } ... }
```

When providing an addition class for an inner class, a corresponding outer addition class must also be provided, as `B` and `A` in the example above shows, respectively.

Returning now to the memento pattern described above, we can utilize our pattern implementation to provide memento functionality for the `Line` class as shown below:

```
inst MementoPattern with Originator => Line
   { State => LineState } ;
inst Drawing ; // or explicitly: inst Drawing with Line => Line

class Line adds {
  protected State GetState() {
    return new LineState()
       {{ x1 = this.x1; x2 = this.x2; ... }};
  }
  public void SetMemento(Memento m) {
     this.x1 = m.state.x1; ...
  }

  protected class LineState adds {
    protected int x1, x2, y1, y2;
} }
```

To utilize the new capabilities added to `Line`, the `Screen` may call `CreateMemento` at fitting points (for instance when a change has been made, utilizing the Observer pattern presented in the next section), and support undo through `SetMemento`.

9.3.4 Aspect-Oriented Programming and the Observer Pattern

The observer pattern is a design pattern with two roles, the *subject* and the *observer*. Each subject maintains a list of observers that are interested in being notified when certain (yet unspecified) changes occur in the subject. An observer may choose to observe one or more subjects at any given time. Such a notification mechanism is a good example of some of the strengths of aspect-oriented programming (AOP) [8].

AOP revolves around the notion of *crosscutting concerns*, i.e. concerns that are not easily captured in any one part of the chosen class hierarchy of an OO application. Such concerns may in (traditional) AOP be defined within a construct known as an *aspect*. The aspect contains code, called *advice*, that is *weaved* in at locations in the base code specified by *pointcuts*.

Usage of the inherent merge capabilities of PT may in itself be seen as a form of code weaving, so in that sense the addition of more AOP-like constructs to strengthen its capabilities in this regard seems like a natural path to follow. Furthermore, mechanisms that bear some resemblance to PT (such as e.g. [16]) have incorporated AOP to great effect.

An important goal for PT is to retain static safety. Thus we seek to avoid some of the issues posed by the *fragile pointcut problem* [22], and shall therefore consider a

somewhat restricted version of common AOP concepts. We believe that paired with PT this may still provide constructs that are sufficiently powerful and expressive for a lot of purposes.

In our approach, aspects are realized not as separate entities (neither at runtime nor compile-time), but rather as pointcuts and advice defined as members of template classes, where a pointcut can only refer to directly visible methods, fields, etc. Pointcuts and advice can be redefined in addition classes and subclasses in the same way as methods. This entails that the possible changes to the base program through the weaving of a pointcut/advice will be limited to the classes of the template, and the addition classes (and their subclasses) of the corresponding instantiation. Thus, quantification using wild-cards as in e.g. AspectJ is not used in our AOP extension of PT. Thereby, we can control the typing of the program to quite another extent. See syntax and example below.

Pointcuts are declared inside classes according to the following EBNF grammar sketch, where terms in quotes are terminals and the unquoted ones are non-terminals. Terms enclosed in curly brackets may be omitted or repeated, while terms in square brackets may be present one time or not at all. Productions that are equal to their Java equivalents are left out for brevity, and things enclosed in << and >> should be understood as "pseudo-EBNF" in place of parts left out. The abbreviation `pc` is used for *pointcut*:

```
pc_decl ::= { modifier } "pointcut"
 ( qualified_identifier | "void" | "*" )
 identifier "(" [ <<parameter list>> ] ")"
 ( "{" pc_expr "}" | ";" )

pc_expr ::=
 ( "call" | "execution" | "get" | "set" ) "(" identifier ")"
 | pc_expr "&&" pc_expr
 | pc_expr "||" pc_expr | "(" pc_expr ")"
```

Likewise, an advise has the following syntax:

```
advice_declaration ::=
  { modifier } "advice" identifier
  ( "before" | "after" | "around" ) identifier
  "{" { <<valid statement>> } "}"
```

To illustrate the semantics of the language, we consider the following example:

```
template T {
  class A {
    A m1(A a) { ... }
    pointcut A pc1 (..) { call(m1) }
} }
```

The class `A` contains a pointcut that matches calls to the method `m1`. The template may be instantiated as shown below:

```
inst T with A => B;
class B adds {
  pointcut B pc1(..) { call(m1) || call(m2) }
  advice afterM after pc1 { ... }
  B m2() { ... }
}
```

Here, `A` is re-typed to `B`, and `B` adds a method and an advice, and overrides the pointcut `pc1`. Note that even though `pc1` in `T` refers to the type `A`, the pointcut will after instantiation refer to `B`, and continue to match method calls to `m1`. This is made possible by the fact that pointcuts are not string patterns, but actual bindings, as mentioned above.

Returning now to the Observer pattern, we consider again our example package for drawing lines on a screen. When a line changes its length or position, the observing screen(s) should be updated to reflect the changes. We would like this logic to be abstracted out of the concrete `Screen` and `Line` classes, so that it can be reused for other manifestations of this particular problem.

Utilizing package templates, we could implement the pattern with the following code[3]:

```
template ObserverProtocol {
 public abstract class Observer {
   abstract void notify(Subject changee);
 }
 public abstract class Subject {
  List<Observer> observers = new List<Observer>();

  public void addObserver(Observer o) {observers.add(o);}

  abstract protected pointcut * changed(..);
  protected advice ac after changed {
    foreach(Observer o in observers) { o.notify(this); }
} } }
```

The `Observer` class has only one method, the abstract `notify`, that will have to be implemented at a later stage to make it meaningful to the concrete observers.

The subject class has methods for adding and removing observers from the current subject instance. Furthermore, it defines an abstract pointcut `changed`, that will have to be refined in concrete subjects. The pointcut specifies that any parameters and return types are valid for its (as of yet undefined) corresponding join points with the use of wild-cards "`*`" and "`..`". We can now do the instantiation as follows:

[3]The removeObserver method is trivial and hence omitted for brevity.

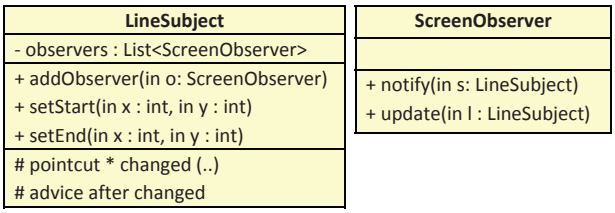

Figure 9.1: The resulting classes from the merge of the ObserverProtocol and Drawing templates

```
inst Drawing with Screen => ScreenObserver, Line => LineSubject;
inst ObserverProtocol with Observer => ScreenObserver,
  Subject => LineSubject;
```

`ScreenObserver` and `LineSubject` become a merge of `Observer` and `Screen`, and `Subject` and `Line`, respectively. Furthermore, we make use of addition clauses to concretize the abstract members of the template classes. These clauses are only needed in order to concretize what was left as abstract in the respective templates.

```
class ScreenObserver adds {
 void notify(LineSubject l) { update(l); }
}
class LineSubject adds {
 pointcut * changed(..) {call(setStart) || call(setEnd)}
}
```

The two resulting classes are shown graphically in Figure 9.1 (where a fourth compartment has been added to the UML class representation to hold AOP members). Note that that a re-typing has occurred, such that for instance the `addObserver` method now takes a `ScreenObserver` as its only parameter. Similarly, the `notify` method now takes a `LineSubject` parameter. Due to the re-typing done by the PT mechanism, no casts are required, and all the code is statically known to be type safe.

Admittedly, this example is a simple instance of the more general class of subject/observer problems. However, the approach presented above applies (with minor variations) to the general case as well. For a more detailed exposition, with multiple interacting subjects and observers, the interested reader is referred to [3].

9.4 Related Work

The original description of the design patterns by the so-called "Gang of Four" (GoF) found in [9] comes with example implementations in C++. For the Singleton pattern, suggestions for reusable implementations are provided (for instance by relying on a

registry in which singleton subclasses need to register themselves), but none of these provide the ease of reuse that the PT version provides.

For the Memento pattern, the GoF implementation makes use of C++'s `friend` construct, to allow only the originator access to the internals of the memento's state. Still, it does not provide the plugability that the PT version does.

The Observer implementation makes use of C++'s support for multiple inheritance to provide subject and observer classes for the pattern. Since we are targeting Java in this example, such inheritance hierarchies cannot be used (but we can, in this case, achieve the same end result, and more, with class merging). The GoF version of Observer also makes use of a special "trick" in which the observer knows the identity of the (single) subject for changes in which it is interested. Because of this they can avoid having to do a type cast that would otherwise have been necessary in the general case.

In *Design Pattern Implementation in Java and AspectJ* [10], Hannemann and Kiczales implement the design patters from the GoF in AspectJ, and contrast these implementations to their potential pure Java counterparts. The Singleton pattern is realized with an *around advice*, that wraps the constructor of the class in question to return a single instance every time it is called.

The Memento pattern may take advantage of AspectJ's possibilities for superimposing functionality onto existing classes. However, a general reusable implementation would be hard, if not impossible, to make without sacrificing type safety.

In the Observer implementation, the pattern is handled by one single abstract aspect, the `ObserverProtocol`. This aspect will exist as an entity at runtime, and contains a global map of observers to subjects. This is in contrast to the PT version, in which there is no notion of the aspect as a separate entity at runtime, but rather that the roles of the aspect are imposed on (and localized in) the classes that are to play the respective roles.

The AspectJ aspect defines empty "marker" interfaces, that conceptually declare the existence of the two roles of the pattern. However, since the interfaces are empty, in contrast to the PT version the aspect does not make it explicit which operations/behaviors belong to either participant in the pattern. In order to update the observer(s) when a change has occurred, a runtime cast must be made from the interface `Observer` to the class `Screen`, since the interface is empty and hence has no knowledge of the `Screen`'s `display` method.

In PT, the concretization happens through the `inst` statement (with optional addition classes for e.g. concretizing abstract members). The re-typing may in most cases eliminate the need for casts completely. The fact that pointcuts are not explicitly bound to join points, means that the AspectJ version is more prone to errors stemming from faulty pointcuts, while the PT version sacrifices some flexibility for a greater degree of pointcut safety.

In *Conquering Aspects With Caesar* [16], Mezini and Ostermann use the Observer pattern as a running example to address two main points with regard to the AspectJ implementation [10] discussed above; (I) the need for expressing aspects not as a monolithic entity, but rather as a set of interrelated, interacting modules, and (II) the need for flexible and reusable aspect bindings and implementations.

With respect to (I), Caesar achieves this through so-called *aspect collaboration interfaces* (ACIs). The ACIs are hierarchical, and may contain several (related) subinterfaces. Each interface may describe a set of provided and/or required operations. The provided operations must be realized by implementers of the interface (e.g. the `addObserver` method must be implemented by an implementation of the `Subject` interface). This brings us over to point (II).

The required operations of an ACI must be implemented by a *binding* (which is separate from the implementation discussed in the previous paragraph), that binds the ACI to classes in the base program. Bindings may be defined independently of ACIs and their implementations, offering greater flexibility than the AspectJ counterpart.

In the PT example, the template is the rough equivalent of both an ACI and its implementation. We could, however, have used interfaces for the different pattern roles to separate actual implementation from public interface, but we are not obliged to.

A CaesarJ binding can be compared to the `inst` statement (with optional addition classes) in PT, in the sense that required/abstract members will (or at least might) be concretized, and roles will be mapped to base program classes.

In *Explicit Programming* [4], the authors present an orthogonal approach in which design concepts (including patterns) may be added directly to the target *language* (i.e. Java). This allows, in principle, any pattern to be represented as a first class language construct, and this is exemplified with the Flyweight pattern. These constructs are implemented as transformations based on *textual* templates.

BETA [14, 15], gbeta [7] and J& (pronounced "jet") [18] are systems that in many ways are similar to each other, and in some respects can achieve similar end results to those of PT. A common property of all of them (except PT, that is) is that they utilize virtual classes (as introduced by BETA) to enable specialization and adaption of hierarchies of related classes. A distinguishing feature of PT compared to these three, is that it allows the developer to quite freely merge previously unrelated classes, and have the corresponding typed references re-typed accordingly.

9.5 Concluding Remarks and Future Work

We have extended the package template mechanism with a couple of desirable extensions, and shown how this can be used to implement components that may be plugged into an existing architecture with a minimal amount of “glue” required from the developer. Though exemplified here by three relatively simple design patterns, we believe that the constructs are usable for a broad range of problems, and that they may aid the developer in his/her continued strive for reusability and separation of concerns.

The AOP constructs introduced here should not be viewed as set in stone, as we acknowledge that the assumptions on which our restricted set of constructs are based may need some tweaking. An interesting topic in that respect is to investigate further the interplay between PT and AOP through experimenting with different degrees of AOP complexity with regard to e.g. pointcut scope and expressiveness, and see how

different points along this axis pair up with inherent PT mechanisms such as re-typing and merging.

Another possible direction would be to see how far one could get with support for merging nested classes compared to what is attainable with the different approaches to virtual classes.

Furthermore, it would be clearly interesting to try to validate the applicability and usefulness of the constructs we have here, with regard to a broader set of real-world problems.

Acknowledgements

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30). We would like to thank the anonymous reviewers for their comments, and Fredrik Sørensen and Birger Møller-Pedersen for valuable input and participation in work upon which this paper builds.

Bibliography

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] AspectJ Team. The AspectJ programming guide, 2003.

[3] E. W. Axelsen, F. Sørensen, and S. Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 37–42, New York, NY, USA, 2009. ACM.

[4] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *AOSD '02: Proc. 1st int. conf. on AOSD*, pages 10–18, New York, NY, USA, 2002. ACM.

[5] A. Colyer. AspectJ. In *AOSD*, pages 123–143. Addison-Wesley, 2005.

[6] Ecma International. *Standard ECMA-334 C# Language Specification*, 4th edition, 2006.

[7] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[8] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *AOSD*, pages 21–31. Addison-Wesley, 2005.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173, 2002.

[11] Java platform, standard edition 6 api specification.

[12] S. Krogdahl. Generic packages and expandable classes. Technical Report 298, Department of Informatics, University of Oslo, 2001.

[13] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. To appear in the Journal of Object Technology (available now from `http://home.ifi.uio.no/~steinkr/papers/`), 2009.

[14] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[15] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[16] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD '03*, pages 90–99, New York, 2003. ACM.

[17] Microsoft .NET Framework class library. URL: `http://msdn2.microsoft.com/en-us/library/ms229335.aspx`.

[18] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, 2006. ACM.

[19] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121 – 145, 2008.

[20] B. Pugh. The Java memory model. URL: `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[21] J. Skeet. Implementing the singleton pattern in c#. URL: `http://www.yoda.arachsys.com/csharp/singleton.html`.

[22] M. Störzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, Berlin, Germany, September 2004.

Chapter 10

Paper III: Groovy Package Templates — Supporting Reuse and Runtime Adaption of Class Hierarchies

Authors. Eyvind W. Axelsen and Stein Krogdahl.

Publication. Proceedings of the 5th Symposium on Dynamic Languages (DLS), 2009.

Abstract. We show how package templates, a modularization mechanism originally developed for statically typed languages like Java and C# , can be applied to and implemented in a dynamic language like Groovy, by using the language's capabilities for meta-programming. We then consider a set of examples and discuss dynamic PT from the viewpoints of code modularization and reuse, and dynamic adaption of classes at runtime.

Categories and Subject Descriptors. D.3 **[Software]**: Programming Languages; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Classes and objects*; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Modules, packages*

General Terms. Languages, Design.

Keywords. OOP, Modularization, Dynamic Languages, Templates.

10.1 Introduction

The need for flexible mechanisms for code organization, modularization and reuse is well-known in the domain of software development. Package templates (PT) have been introduced [21] as one approach to this for statically typed languages like C# [8] and Java [13]. Given the promising preliminary results of applying this mechanism to example problems in Java [2, 32], it is an interesting endeavor to examine how this mechanism fits with other languages and paradigms. In this paper, we will examine package templates applied to the dynamic language Groovy [15].

The dynamic nature of the Groovy language opens up several new possibilities for making use of package templates. It also provides an excellent vehicle for studying and experimenting with mechanisms for structuring and reuse of code beyond traditional classes and packages.

In keeping with its dynamic nature, Groovy supports certain modifications of existing classes at runtime. This is a feature that (to varying degrees) is also supported by several other dynamic languages, such as Smalltalk [12], Ruby [29], Python [28], etc. Several different mechanisms may be employed to enable such modifications, such as open classes [5], meta-programming, runtime mixins [4] etc. These mechanisms may be used to achieve things that are difficult or even impossible to accomplish with a static approach. However, with such power comes a set of pitfalls, that in certain scenarios may be significant. We shall explore how dynamic instantiations of package templates may be employed to combine a powerful modularization technique with expressive runtime constructs, and discuss how this may alleviate some of the problems that the other approaches come with.

The Groovy language is in itself an interesting and compelling platform for programming language experimentation from several points of view, and while we evaluated several other languages, its unique set of features contributed heavily to our selecting it as the target language for prototyping our mechanism. To begin with, it compiles to byte code and runs on any standard Java Virtual Machine [22]. This means that the availability of libraries is at least as good as for Java, right out of the box. Also, the syntax, which to a certain extent follows that of Java, should be pleasingly familiar to many developers.

Furthermore, Groovy supports somewhat advanced concepts, such as the Meta-Object Protocol (MOP) [19] and abstract syntax tree (AST) transformations, that are not found in the majority of other languages. We will make heavy use of these features when adding support for package templates to Groovy.

The contributions of this paper are:

- We show how package templates can be beneficial to dynamic languages, and how the nature of such languages can be exploited to make package templates even more flexible. Furthermore, we show how runtime adaption of classes can be done in a comparatively clean and safe manner with PT. We demonstrate the utility of these concepts by a set of examples.

- We describe a prototype/proof-of-concept implementation of package templates

for Groovy, supplied in the form of pluggable libraries that require no change to the source code of neither the Groovy libraries nor the compiler. This allows interested parties to modify the semantics of our implementation with relative ease, since it is entirely built from standard Groovy classes.

The remainder of this article is structured as follows: In Section 10.2, we briefly describe the main concepts of PT, and the distinguishing features of the Groovy language. Section 10.3 describes the main concepts of dynamic PT, while a set of examples is presented in Section 10.4. The implementation of the mechanism is briefly discussed in Section 10.5, and Section 10.6 deals with related work. Section 10.7 concludes this article, and touches on possible ideas for future work.

10.2 Background

10.2.1 Package Templates at a Glance

The package template (PT) mechanism targets the development of reusable collections of interdependent classes. A main idea is that the template mechanism should be flexible enough to allow e.g. framework developers to write quite general templates, and then to provide the application developer with powerful mechanisms for tailoring the templates of the framework to his or her specific needs. An important point in that regard is that such tailoring should be unintrusive with respect to other parts of the application that might be using the same framework.

A package template looks much like a regular package in the target language, with the most notable exception being a `template` construct enclosing classes and interfaces. Figure 10.1 shows two example templates: template `T` with three classes, `A`, `B`, and `Aa`, where the latter inherits from `A`, and template `U` with only one class, `C`. To model package templates, we will in this paper use a graphical notation where UML classes and UML interfaces are enclosed by a bordered gradiently colored box with an underlined "template [name]" header, representing a template.

Package templates must be *instantiated* before use, and access to the contents of the instantiated template will thus be provided from within the instantiating scope. Each instantiation of a given template is completely independent of any previous instantiations. In the following, we shall use the term *package* in a broad sense, meaning a collection of (possibly related) classes and interfaces that is internally consistent, but not necessarily implemented as a Java or Groovy package as such. Using this terminology, each template instantiation will result in a new package being created.

Upon instantiation, the template may be customized according to its usage, and classes from different template instantiations (of the same or different templates) may be merged to form one new class. For instance, we may choose to instantiate the templates `T` and `U` from Figure 10.1, and to merge the classes `Aa` from `T` and `C` from `U` together, and give the merged class the name `AC`. Furthermore, we might add members to any of the classes, e.g. a new method `m4` to `B` and a new field `field3` to `A`. Template classes and interfaces may also be renamed, and templates may have (con-

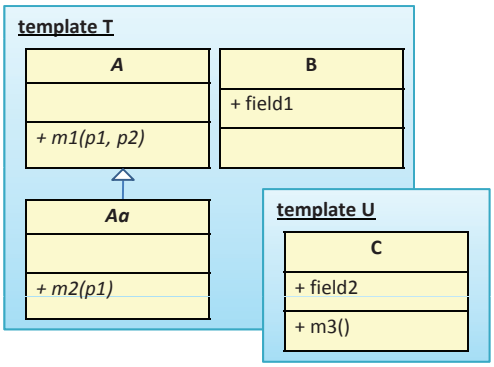

Figure 10.1: Two example templates, `T` and `U`.

strained) type parameters (though the latter is of little interest in the dynamically typed setting we shall stay within in the rest of this paper). The resulting classes now available in the instantiating scope are shown in Figure 10.2. Note that the classes in the instantiated template have no relation to their template counterparts, neither through inheritance relationships nor otherwise. Correspondingly, there are also no relations between classes from different instantiations of the same template(s).

10.2.2 Groovy

Groovy is a dynamic language that runs on the Java Virtual Machine. It can hence integrate nicely with Java, and to a certain extent it provides a familiar Java-esque syntax. The language rests on a purely object oriented foundation, and provides support for several attractive language concepts.

Both static and dynamic typing of variables and method signatures is supported in Groovy. However, there is presently no performance to be gained in Groovy by using static typing as opposed to dynamic (and in many cases it may actually be slower). Furthermore, Groovy cannot detect e.g. invalid calls to statically typed methods at compile time (since a fitting method may be added to the object in question at runtime) [16]. We will therefore, and for other reasons mentioned earlier, not consider the static typing features of Groovy in this paper, and instead focus solely on dynamically typed references, methods and properties defined by using the keyword `def`.

Dynamic typing aside, one thing in Groovy that is resolved entirely at compile-time is the given class from which to create a new object (e.g. as in `new A()`). That is, if a referenced class `A` cannot be statically determined to exist within imported packages and with correct accessibility modifiers, a compile-time error will occur. This will pose problems for our implementation, since the dynamic instantiation of a package template will make new classes available at runtime. We shall look closer at how we resolve this when discussing our implementation in Section 10.5.

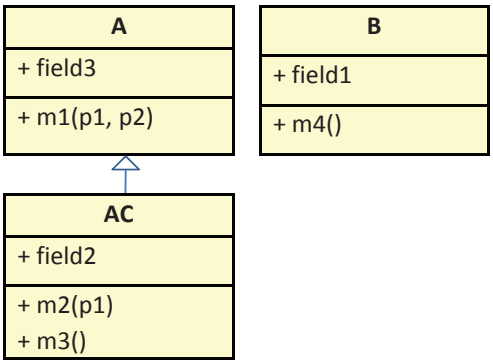

Figure 10.2: The resulting classes from the instantiation of templates `T` and `U`, with `Aa` and `C` merged. Members have been added to `A` and `B`.

A runtime meta-object protocol (MOP) is supported by the Groovy language. In accordance with the MOP, every object has an associated object instance of its meta-class. This object may be replaced by a version defined by the programmer, and this allows the semantics of method invocation and property getters and setters to be changed on a per-object or per-class basis. Also, new members may be added to classes and objects at runtime using the meta-class.

Further tailoring of the language's semantics may be done using compile-time transformations on the abstract syntax tree. AST transformations may be defined by the developer to transform or replace parts of the syntax tree during different phases of the compilation process, such as parsing, semantic analysis, code generation etc. Such transformations may be global, or local pertaining to certain targets that typically may be Java-like annotations on source code elements.

Closures are first class citizens of Groovy, and units of code may as such be passed to and from methods.[1] Closures are hence also objects, and have their own meta-class instance that may be replaced by the programmer. They may also have a *delegate*, to which method calls and field accesses are dispatched. The delegate is, as the meta-class, replaceable by the developer. We will make good use of these features when defining our framework for instantiating templates.

10.3 Dynamic Package Templates

We shall now describe package templates for dynamic languages, exemplified by an implementation for the Groovy language. The two main points that motivate such an

[1]While Groovy closures are not actually closures in the formal sense of the word, we shall in this paper use the term *closure* to refer to closures as the syntactic constructs that define subclasses of `groovy.lang.Closure`, and instances thereof.

implementation are

- to make full use of the freedom that the typeless nature of many dynamic languages provide, in order to achieve greater flexibility when it comes to adaption and composition of templates upon instantiation, and,
- to leverage (and extend) the capabilities of PT in a dynamic setting, and provide a powerful framework for coherent reuse and runtime adaption of entire class hierarchies.

Furthermore, as was mentioned in the introduction, using this implementation as a vehicle for studying the properties of package templates as a software modularization technique without regard for typing issues comes as an added bonus.

10.3.1 Writing Templates

A package template in Groovy is written as follows:

```
template T {
   // classes and interfaces here
}
```

The contents of the template are ordinary Groovy classes or interfaces. These types, referred to as *template members*, may refer to each other and take part in inheritance hierarchies within the template. A template class is not allowed to have inheritance relationships to, or references to, classes outside of the template, with the exception of classes of templates that are instantiated within the current template. Templates need to be semantically (and, of course, syntactically) valid in their own right, and may as such be checked for validity according to the rules of the target language (in this case Groovy) independently of their potential usage.

None of the classes or interfaces defined in the template are available to the rest of the program before the template itself is instantiated (that is, the class loader does not see any of the members of a template before an instantiation of the template in question is performed).

10.3.2 Instantiating Templates

In the Java version of PT, template instantiation is performed statically at compile time, providing access to template classes and interfaces to every class and interface within the instantiating package. Instantiation can only be performed at the outermost level of a file, i.e. at the same level as any `import` statements.

In this dynamic version, the Groovy PT framework provides a set of methods[2] that may be invoked at runtime in order to instantiate templates. The motivation for studying dynamic instantiations as well as static, is that it is easy to imagine situations in which it cannot be known until runtime which version of a specific functionality is

[2]Or, if you will, a tiny internal DSL

needed, or if certain modifications should be applied to an entire hierarchy of classes or not. Supporting the stance that this is actually useful for solving real world problems is the fact that many popular dynamic languages, such as e.g. Python, Ruby and of course Groovy, provide support for class modifications at runtime, and popular frameworks such as Rails [30] and Grails [14] make heavy use of this. We shall see examples of how this can be utilized in dynamic PT in the examples in Section 10.4.

A package template may be instantiated at run-time by utilizing the `instantiate` method of the supplied `PT` class, as shown in the following construct:

```
PT.instantiate { template T }
```
[3]

This statement will make *a copy of* all the classes and interfaces defined in the template `T` available to the caller of the static `instantiate` method, preserving potential type hierarchies as defined in the template.

However, instantiations may also be more elaborate. The `instantiate` method takes a Groovy closure as its sole argument, and this closure may in turn contain several calls to the `template` method. The `template` method takes a template name (such as `T` in the example above), and optionally a specification (in the form of a closure) of *adaptations*, such as mappings to new names etc.

If multiple templates are listed in the argument provided to `instantiate`, this will lead to one single instantiation of these templates, forming one single new package. Taking the example from Section 10.2.1 (as illustrated in Figures 10.1 and 10.2), we would employ the following call to merge the classes `Aa` and `C` (we will get back to how to add fields and methods, as in the example, in Section 10.3.3):

```
def INST_TU = PT.instantiate {
    template T { map Aa, AC }
    template U { map C,  AC }
  }
```

Here, the classes `Aa` and `C` are both renamed (or *mapped*) to `AC` (using the `map` method), and they will thereby be merged (more on that in Section 10.3.4). A new, identical (from a textual point of view), copy of each of the classes `A` and `B` will be available to the instantiating context, while the template classes `Aa` and `C` will not be available as individual classes, but only as their merge.

In the preceding paragraphs, we have not discussed the issue of the scope of instantiated templates. In a statically typed scenario, where template instantiations are performed at compile-time, the scope is always the instantiating package. However, when we are looking at dynamic instantiations, which may be performed at arbitrary points at runtime, the issue of instantiation scope becomes important.

[3] The syntax of Groovy is quite flexible, allowing the developer to omit parenthesis and semicolons where desirable. Furthermore, we massage the resulting expression a little at runtime, to allow the developer to omit commas and quotation marks. The instantiate statement could hence also be written in a more elaborate (and perhaps more immediately familiar) way: `PT.instantiate({ template("T") });`.

When dynamically instantiating templates (either from within methods or as part of the static initialization phase of a class), a variable holding the resulting package needs to be explicitly specified, in order to provide the correct scope for the new package. E.g., given templates `U` and `V`, with template classes `A` and `B`, respectively, one could for instance have the following instantiation:[4]

```
void f() {
   def INST_UV
   if(someCondition) {
      INST_UV = PT.instantiate {
         template U { map A, Z } }
   } else {
      INST_UV = PT.instantiate {
         template V { map B, Z } }
   }

   def z = new INST_UV.Z()
}
```

This allows the runtime boolean condition `someCondition` to potentially affect an entire hierarchy of classes, providing specific adaptions to classes which may be used in subsequent stages of the application. The scope of the instantiated class hierarchy is well defined by the scope of `INST_UV` (which follows the normal scoping rules for variables in the language). New objects may be created from the instantiated template classes by referring to them with their qualified name, e.g. as shown above as `INST_UV.Z`.

Given a mechanism such as this, it is now quite possible (yet perhaps somewhat contrived) to write code that performs the same instantiation multiple times, e.g. within a loop such as the code snippet below.

```
for(;;) {
  INST_T = PT.instantiate { template T }
  a = new INST_T.A()
}
```

For each iteration of this loop, a new template instantiation would be performed, and the class `INST_T.A` in a given iteration would be different (though equal in structure) to that of the previous or next iteration. Since there may be multiple instantiations of any given template (or sets of such), each instantiated template class is given a reference to its instantiation, so that other classes from its own instantiation will take precedence over external classes when new objects are to be created.

Arguably, one would often want the classes and interfaces of a template to be available to every class in a given package, without having to perform several instantiations

[4]For clarity, we shall name all variables holding instantiated templates with the prefix `INST_` followed by the name of the template(s) or a derivation thereof. This is however only a convention, and not something that is enforced by our implementation.

of the same template, in the same way as is the case for the static version of PT. The recommended way of achieving this in dynamic PT is to create a class that instantiates the template when it is loaded by the class loader. This class may then be statically imported for easy access in other files. The example below illustrates this:

```
class TemplateT {
   static def INST_T = PT.instantiate { template T }
}
```

Given the class above, other classes may utilize the instantiation of `T` by importing `TemplateT`:

```
import static TemplateT.*
class C {
   def m () { def a = new INST_T.A() }
}
```

10.3.3 Additions

When instantiating a template, template classes may be customized to the current usage by so-called addition classes. Such a class specifies method overrides and new members, and can be written as an ordinary class, with an additional `@Addition` annotation. For instance, consider the following template, and a corresponding instantiation of it:

```
template T {
   class C { def f() { /* something */ } }
}

def INST_T = PT.instantiate { template T }
```

The virtual method `f` in `C` may be overridden in an addition class, and new members may be added to the class, e.g. as follows:

```
@Addition class C {
   def f() { */ something else */ }
   def g = ""
}
```

The instantiated class `C` will have the added members and overrides of the addition class `C` (based on their sharing a name). Stated explicitly, the class `C` will after the instantiation have the method `f`, with the implementation given in the addition class, and a field `g` which is initialized to the empty string.

Once more, looking back to the example in Figures 10.1 and 10.2, the addition classes for `A` and `B` can be written in a straightforward manner, as follows:

```
@Addition class A { def field3 }
@Addition class B { def m4() { /* impl. here */ } }
```

The `@Addition` annotation ensures (through a local AST transformation defined by the PT framework) that objects may not be created from its belonging class until a template instantiation that targets this class has been made. It also serves as a marker interface for the implementation that needs to record certain pieces of meta-data about this class during the compilation process.

Keeping with tradition in Groovy, a method may also be defined by providing a closure containing its new implementation, and this closure may be used to provide a suitable override directly when instantiating the template, e.g. as follows:

```
def INST_T = PT.instantiate {
    template T {
      map C {
        override f { /* something else */ }
 } } }
```

If a template class is declared to be abstract, it may be made concrete upon instantiation by utilizing a non-abstract addition class, that provides concrete implementations of all abstract class members. This has turned out to be especially useful for implementing templates that represent some form of abstract protocol, where parts of the interaction can be specified in a totally generic way, while other parts need to be specified by the domain classes implementing the protocol. We shall see an example of this when looking at the Observer design pattern in Section 10.4.2.

10.3.4 Merging, Renaming and Exclusion

It seems to be a well established recognition within the OO community that traditional single inheritance alone is too limited to capture scenarios where complex reuse of code is needed. Many real-life situations actually require (or would benefit from) the reuse of code from more than one distinct entity. Multiple inheritance is one way to achieve this, but that comes with a host of well-known problems of both technical and conceptual nature. Mechanisms like traits [31] and mixins [4] aim to solve this problem by creating conceptually simple units of code intended only for code reuse, and not for object creation. PT offers a different approach to this kind of reuse, in which classes from independent template instantiations may be merged together, while preserving type hierarchies and avoiding multiple inheritance (we will get back to the latter).

The merge of two classes `A` and `B`, is defined as taking the discriminated union of the set of all the members (methods, fields and constructors with associated meta-data[5]) from `A` and the set of all the members of `B`. Members with the same name and signature stemming from different classes are hence not considered equal, and will therefore appear twice in the resulting set. This is defined as a conflict. For instance would the following not be allowed, and would lead to a runtime exception if not explicitly handled by the programmer:

[5]Inner classes are not mentioned because they are presently not supported in Groovy

```
template T { class C { def f(arg1) { ... } } }
template U { class D { def f(arg2) { ... } } }

def INST_TU = PT.instantiate {
    template T { map C, CD }
    template U { map D, CD }
  }
```

Here, the classes `C` and `D` are merged under the common name `CD`. Since they both define a method `f` with a single argument, there is a conflict. Every such conflict should be resolved manually by the developer. In static PT, one typical way to resolve conflicts, is to rename members (statically) until there are no more conflicts (e.g. in the example above, `C.f` could be renamed to `C.g`). Since the semantic analysis will bind every method call to a particular signature, renaming may safely be done (of both the member itself and all references to it). Given the dynamic nature of both the type system and method call resolution/dispatch in Groovy, this is not possible. With statements like e.g. `def f` followed by a call such as `f.m()`, it would be impossible to safely rename `m`, since the type of `f` can not be determined statically.

One general approach to renaming could be to keep a list (or some other fitting data structure) of all renamed members inside a class, and perform dynamic dispatch through this mechanism when in a *message not understood* situation. This works for renames of members from a single class, but it will not be of any help when there are merge conflicts involved, since it cannot be known if the call/field reference was intended for the renamed method/field or for the originally conflicting one that was left untouched in the merge.

Given these issues, it seems clear that renames cannot be used to resolve merge conflicts. Instead, a possible option would be to exclude one of the conflicting members, so that all calls/references would be routed to the remaining member. Exclusion may be done with e.g. the following instantiation (referring to the example templates `T` and `U` defined above):

```
def INST_TU = PT.instantiate {
    template T { map C, CD { exclude f } }
    template U { map D, CD }
  }
```

If one excludes members that are not the source of a merge conflict, there is always the possibility that these members are being referenced other places in the template. Since this cannot be known in advance (due to the dynamic typing issues discussed above), we have essentially two options; either to disallow exclusions that are not part of conflict resolution, or to allow it and let the developer deal with it. We have opted for the latter, and allow exclusion of members upon template instantiation[6]. One exception to this rule that is probably reasonable to make (though we have not yet implemented it),

[6]Allowing the exclusion of members, though potentially unsafe, is also the approach taken by the designers of Traits.

is to disallow the exclusion of members that are part of an interface that the class in question is declared to implement. To allow exclusion in such a scenario, the member would then first have to be excluded from the template interface, and then from the implementing class(es).

Another way to resolve a merge conflict when the conflicting members are methods, is to provide an override in an addition class. This override will in that case take precedence over all methods with equal signature stemming from template classes, and may access the original implementations through `super` calls, qualified with the name of the respective template class to which it belongs, as in e.g. `super[C].m()`[7].

The merge of two independent classes could implicitly lead to multiple inheritance, which we do not allow. Thus, some restrictions on allowable merges are needed. The first restriction is that we forbid template classes to have superclasses defined outside the template. This may seem drastic, but one should remember that template classes can still freely implement interfaces defined outside the template, and this will to a large extent make up for the inconvenience introduced by the above superclass rule. Superclass relationships between classes inside a template are of course allowed. The second rule is that if, in an instantiation, two or more template classes are merged, then the superclasses they have (which, if they exist, are also template classes) must also be merged in the same instantiation. We also have to add the requirement that a merge must not result in a cyclic superclass-structure.

After a merge or a rename has been done, statements inside the instantiated template classes that create new objects (e.g. of the form `new C(...)`) need to processed by the PT framework so that they refer to the new name of the merged and/or renamed classes.

10.4 Examples

In this section, we will look at some examples in order to further motivate and explore the dynamic PT mechanism. We shall look at some properties that apply to PT in general, and some that are only attainable in the dynamic version.

10.4.1 Lists and Matrices

To begin with, we will here look at how a simple standard linked list implementation (adapted from an example in [21]) can be implemented and utilized with dynamic PT. First, we define a template containing classes for the list itself, and for elements of such lists, as shown in the following code:

[7] In some situations, particularly when instantiating the same template twice, a more elaborate way of referring to the original template class of a method may be needed.

```
template Lists {
   class List {
       def first, last
       def add(element) {
          if(first == null)
             first = element
          else
             last.next = element
          last = element
       }
       ... methods for removing etc ...
   }
   class Element { def next }
}
```

The `Element` class is of course not all that useful as is, and the intention is that upon instantiation, this class should have an addition class specifying additional members corresponding to the given problem domain, or be merged with a class from a different template. For instance, we may create a list of students by merging the `Element` class with a `Person` class as follows:

```
template Persons {
   class Person {
      def lastName, firstName
      ...
   }
}

def INST_PList = PT.instantiate {
   template Lists { map Element, Student }
   template Persons { map Person, Student }
  }
```

However, a `Student` class would probably need a few more properties and perhaps some added logic compared to a regular `Person` class. The necessary additions and/or potential overrides can easily be done by creating a corresponding addition class:

```
@Addition class Student {
   def studentNumber
   def attendingClasses = []
   ...
}
```

It is also possible to merge classes from the sample template, when this template is instantiated more than once. Below is an example of this, where we show how we may

now use our `Lists` template to create a sparse matrix by instantiating this template twice, and mapping the respective classes accordingly (remember that every template instantiation is independent of any other instantiation of a given template).

```
template Matrix {
  PT.instantiate {
     template Lists {
        map List, SparseMatrix
        map Element, Column
     }
     template Lists {
        map List, Column
        map Element, Item
     }
  }

  @Addition class Column { def cNo }
  @Addition class Item { def rNo }

  @Addition class SparseMatrix {
     Item getItem(columnNo, rowNo) {
        def col = first
        while (col!=null && col.cNo!=columnNo){
           col = col.next
           ... and so on ...
        }
     }
  }
}
```

To use the `Matrix` template in a real program, one might now simply instantiate the template and map the `Item` class to a fitting domain abstraction, or create new addition classes to specify additional required state or behavior.

10.4.2 A Reusable Observer Pattern Implementation

The observer pattern is a design pattern [11] with two roles, the subject and the observer. Each subject maintains a list of observers that are interested in being notified when certain (yet unspecified) changes occur in the subject. An observer may choose to observe one or more subjects at any given time. We looked at this pattern in [2], utilizing an aspect oriented extension to Java PT to create a reusable implementation. Here we will look at how the same can be achieved with dynamic PT in Groovy.

To exemplify the use of the pattern, we consider a package for drawing objects on a screen or printing them out on a printer (strongly resemblant of the examples in [17] and [25]). The `Drawing` package is realized as a package template as shown in Figure

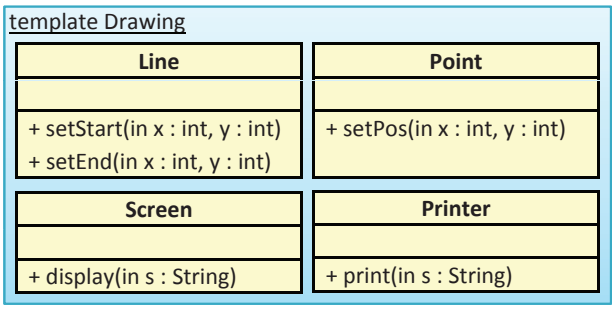

Figure 10.3: The `Drawing` template with two potential subjects and two potential observers

10.3. The `Screen` and `Printer` classes can be considered potential observers, while the `Line` and `Point` potential subjects, respectively. Hence, `Screen`s and `Printer`s should be notified when changes occur in `Line`s or `Point`s.

The Observer pattern can be programmed as a template consisting of two classes, `Subject` and `Observer`, as shown below:

```
template ObserverProtocol {
   public abstract class Observer {
     abstract def notify(changee);
   }

   public abstract class Subject implements
      GroovyInterceptable {
      def observers = []

      def invokeMethod(name, args) {
         def result = this.class.metaClass.
            invokeMethod(this, name, args)
         if(getMutatingMethods().contains(name))
            afterChanged()
         return result
      }

      def addObserver(observer) {
         observers.add(observer)
      }
```

```
        def removeObserver(observer) {
            observers.remove(observer)
        }

        abstract def getMutatingMethods()

        def afterChanged() {
            foreach(observer in observers) {
                observer.notify(this);
            }
        }
    }
}
```

The `Observer` class has only one method, the abstract `notify`, that will have to be implemented at a later stage to make it meaningful to the concrete observers.

The `Subject` class has methods for adding and removing observers from the current subject instance. Furthermore, it defines an abstract method `getMutatingMethods`. This method should return a list describing the methods that mutate the state of the subject in such a way that observers should be notified, and it must clearly be left abstract at this point. The `Subject` also implements the empty `GroovyInterceptable` marker interface, which is a part of the Groovy MOP implementation. This interface signifies to the runtime that method invocation should be routed through the virtual `invokeMethod`. By overriding this method, we may intercept every method invocation, and use the `name` argument to check if it is a call we are interested in. Our overriding `invokeMethod` calls the method that was originally called by using the meta-class, and then checks to see if the method called is a mutating one. If so, it notifies observers by calling the `afterChanged` method.[8]

To apply the observer pattern to the `Drawing` template, we instantiate both templates as follows:

```
def INST_DO = PT.instantiate {
     template Drawing
     template ObserverProtocol {
         map Subject, Line
         map Subject, Point
         map Observer, Screen
         map Observer, Printer
     }
  }
// addition classes here, see below
```

[8]The check for mutating methods is for the purpose of this example a simple one based on name only, but it could of course with relative ease be extended to take the entire signature of the method into consideration, or to allow wild-cards in the vein of e.g. AspectJ [6] etc.

Here, we are using a feature only possible in the dynamically typed version of PT, and that is that the `Subject` class and the `Observer` class are both mapped to more than one target class (i.e. `Line` and `Point`, and `Screen` and `Printer`, respectively) in the merge done upon instantiation. In a statically typed world, this would have introduced typing issues for references that were previously typed as either `Subject` or `Observer`, and would hence not be allowed.

What is missing form the example at this point, is to specify which methods that are considered to mutate state in `Line` and `Point`, and which action should be taken upon call to the `notify` method for `Screen` and `Printer`. For the latter two classes, we would quite simply write addition classes to call the `display` and `print` methods, respectively, from the `notify` method, providing the changed subject (i.e. a `Line` or a `Point` object) as the argument.

For the `Line` and `Point` classes, we assume that every method with a name starting with '`set`' mutates the state, and we specify this by using addition classes:[9]

```
@Addition class Line {
  def getMutatingMethods() {
     return this.metaClass.getMethods().
        findAll { it.name.startsWith("set") }
}
```

The addition class for `Point` would be equal to that of `Line`, with the exception of the class name itself. Having only two subjects, that is probably not much of an issue, but if there were many subjects that all would share the same definition of mutation, duplicating that same code many times would not be desirable. With dynamic PT, we could instead have put the concretization of `getMutatingMethods` into a template class (that is, in a template that is separate from both the `Observer` and the `Drawing` templates), and upon instantiation mapped the same implementation directly to the subject role.

An example program utilizing the instantiated templates could look as follows:

```
def screen = new INST_DO.Screen()
def line = new INST_DO.Line()
line.addObserver(screen)
line.setX(10)
```

The call `line.setX(10)` would now ultimately result in the `screen`'s `display` method being called with `line` as its argument.

Clearly, it is also possible to make an implementation of the Observer pattern in plain Groovy. There are, however, certain worthwhile benefits of using package templates. To begin with, PT allows for one *single* and *coherent* abstraction (the template)

[9] This approach would include the base class methods `setProperty` and `setMetaClass` in the list of mutating methods, which we in a real situation clearly would not want (at least not the latter of the two). This could easily be amended by a little more elaborate approach to the `getMutatingMethods` method, however for the sake of our example and what we want to demonstrate, the simple approach will suffice.

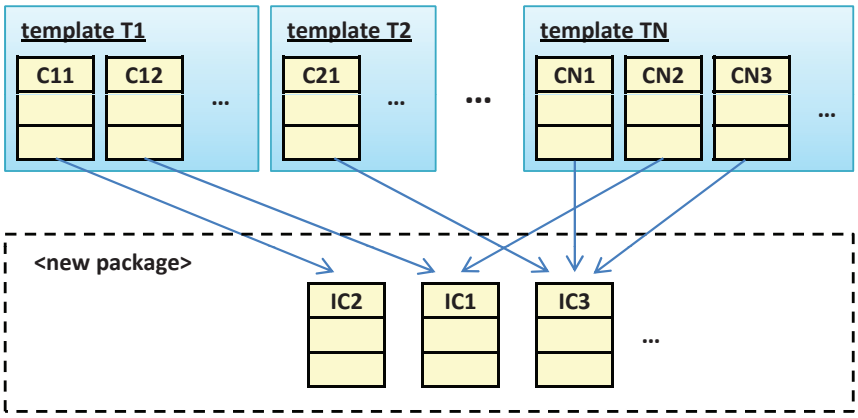

Figure 10.4: A schematic view of how PT can be used to quite freely mix features from different libraries to form one new package. The arrows represent an example instantiation with mapped template classes.

to capture the concept of the pattern. The protocol of interaction between the two roles may be specified in one place, and may be reused as one conceptual unit. Furthermore, PT allows for pluggable reuse of the entire pattern, even when the targeted domain abstractions already are a part of an inheritance hierarchy, without having to resort to multiple inheritance. This does, however, require that the domain abstractions in question are also implemented as templates or addition classes. Last, if the classes of the `Drawing` template were in use other places in the application, adding the `Subject` and `Observer` functionality could be done unintrusively only where needed, and the rest of the application could, if desirable, be left with the original versions of the classes from the template.

If we generalize this example, we see that PT allows the developer to enjoy a great deal of freedom to mix and match several different libraries (templates), and merge required functionality into new classes to form an independent, new, package. Figure 10.4 illustrates this. Package specific adjustments may be made through addition classes, or as parameters to the `map` method.

10.4.3 Adding logging capabilities dynamically

For this example, we shall consider a Groovy application for simulating graph-like structures of various forms that is deployed at a customer's site (and the availability of debugging tools is hence limited). The application is using a graph library in the form of an instantiated package template to model certain aspects of a physical Ethernet network, as well as other structures such as telephone lines etc. A skeleton of the `Graph` template is defined as follows:

```
template Graph {
   class Node {
      def edges = []
      def insertEdge(toNode) {
         def e = new Edge()
         edges.add(e)
         return e
      }
      ...
   }

   class Edge {
      def fromNode, toNode
      def delete() { ... }
      ...
   }
   ...
}
```

The library is used by the rest of the application by utilizing classes with static members holding instantiated template classes, as illustrated in Section 10.3.2:

```
class EthGraph {
   static def INST_EG = PT.instantiate {
       template Graph {
          map Node, Computer
          map Edge, Connection
         }
     }
}
// possible additions to Computer and Connection here

class TelephoneGraph {
   static def INST_TG = PT.instantiate {
       template Graph {
          map Node, Central
          map Edge, Line
        }
     }
}
// possible additions to Central and Line here
```

Consider now a situation in which a certain part of this application is misbehaving, and we suspect that the graph libraries are involved in the bug. In order to debug this, we would like to enable extended logging at the client site, but only for a very limited part

of the application (since extensive logging may be expensive in terms of performance and storage space). To enable logging, we utilize similar techniques as for the observer pattern presented in Section 10.4.2 to create a simple `Logging` template consisting only of a `Logger` class, that overrides the Groovy `invokeMethod` method to provide functionality resembling the aspect-oriented programming construct of *before advice*:

```
template Logging {
  class Logger implements GroovyInterceptable {
     def invokeMethod(name, args) {
        log(name, args) // perform logging
        return this.class.metaClass.
          invokeMethod(this, name, args) // proceed
     }
     ...
  }
}
```

In order to enable logging, we may now create an instantiation of `Logging` that merges the `Logger` class with the classes whose method invocations we wish to log. To minimize the effect on the efficiency of the application, we may choose to only enable logging for a selected class that uses the library. To achieve this, we may change that class so that every object of this class includes its own reference to the instantiated templates. For instance, given a `Simulation` class that uses the `INST_EG` and `INST_TG` instantiations defined above, we may create local variables that hold the instantiated templates, and define methods that may set these variables to instantiations in which logging is merged in at runtime.

```
class Simulation{
   def INST_EG = EthGraph.INST_EG
   def INST_TG = TelephoneGraph.INST_TG

   def setupNetwork() {
      def c1, c2
      c1 = new INST_EG.Computer()
      def wire = c1.insertEdge(c2)
      ...
   }

   def enableEthLogging() {
     INST_EG = PT.instantiate {
       template Graph {
          map Node, Computer
          map Edge, Connection
       }
```

```
        template Logging {
            map Logger, Computer
            map Logger, Connection
           }
        }
    }
    ...
}
```

After the `enableEthLogging` method has been called, every new-statement involving `INST_EG.Computer` and `INST_EG.Connection` will refer to the instantiated package that has the logger class merged with `Computer` and `Connection`. Furthermore, every such subsequent object created by any of these instances will again also have logging enabled, but the rest of the application will remain unaffected. This means that after a log-enabling call, the `Computer` object created in `setupNetwork` will have logging enabled (as its class will be a merge of the `Node` and `Logger` classes). Logging will hence also apply to the `Connection` object that is created by the call to `c1`'s `insertEdge` method in the example above. To turn logging off again, one would simply reset the `INST_EG` variable to its original value (though this would of course not affect any objects that are already created).

This exemplifies how dynamic instantiations can be applied in order to selectively turn features on and off in a application, with a well-defined scope for the changes. One can easily imagine other scenarios where this would be relevant, for instance customizing certain parts of application based on the contents of a license file (resolved at runtime), adapting the GUI to different kinds of devices etc. Dynamic class modifications are supported by many languages, but the pitfalls of using such features is that the modifications may have unanticipated effects on other parts of the application, making the program harder to debug and reason about (since global changes to classes may be done at any point). The main incentive for using PT in such scenarios would therefore be its ability to apply coherent changes to an entire hierarchy in one operation at runtime, and to be able to constrain the scope of such modifications.

10.5 Implementation

Having decided upon Groovy as the target language for an implementation, the inherent flexibility and power of Groovy allowed us to follow a self-imposed restriction: to only make use of the standard features of Groovy itself, and not modify any of its source code. This enables anyone that has Groovy to just plug in our libraries, and experiment with Groovy package templates. As a result of this, the syntax may not be as nice as it could have been (i.e. in contrast to the Java implementation of PT, which employs special keywords for template instantiations and addition classes), but on the other hand it should be familiar to any developer versed in Groovy. It also makes making modifications and extensions to the syntax (and the semantics!) a lot more manageable, effectively making the PT framework open for modification.

The dynamic PT framework is implemented as a Java archive (jar) that can be included in any Groovy solution to provide access to its features. The implementation is mainly divided into three parts; (1) the tiny DSL that allows for expressing template instantiations within Groovy, (2) the actual implementation of such instantiations, and (3) the AST transformations that take place at compile time to allow `new`-statements that include references to as-of-yet not instantiated template classes, and to make sure that addition classes are not used to create objects without having been targeted by a template instantiation.

The first part relies on a set of closures for which the execution is delayed until they are assigned a specific delegate that overrides the `methodMissing` and `propertyMissing` meta-object protocol methods. Utilizing this in a builder-like fashion (resembling e.g. the Groovy `MarkupBuilder`), the framework dynamically builds a tree representing the requested instantiation at runtime. This allows for a quite clean syntax, without the need for modifying Groovy's grammar.

When the tree representing the instantiation has been built, control is handed over to a transformer class that builds a new tree representing the instantiated template. The framework will then use this tree to look for each template that is to be instantiated in a file named `[template name].groovy.template`. If a fitting file is found, its contents are handed over to a simplified template parser, that separates the template classes and interfaces from template instantiations and `import` statements at the outermost level of the template. For each template class or interface, a specialized class loader that records the AST for each such respective element is applied.

The result of a template instantiation is an instance of the provided `InstantiatedTemplate` class. To this class, properties for each instantiated template class are added (at runtime). For instance, for a template containing classes `A` and `B`, the object instance of `InstantiatedTemplate` will have corresponding properties named `A` and `B`. Each such property contains an instance of the `Class` class, that provides access to the instantiated template classes to the rest of the program at runtime. For each instantiated template class, a property `__INST` is added, and set to a reference to the instantiated package of which the class is part. By doing this, we can ensure that classes from the same instantiation will take precedence over potential different instantiations of the same class when new objects are to be created.

As the final part, calls to constructors (e.g. like `new C(...)` or `new INST_T.C(...)`) need to be processed at compile-time. This is because classes and interfaces are resolved statically by Groovy, while dynamic PT allows for dynamic class generation at runtime. Our approach is hence to detect the object instantiations for which the class cannot be statically resolved, and instead inject a runtime call to the `newInstance` method of the instance of the corresponding `Class` class, by applying an AST transformation to the source tree. Furthermore, classes with the `@Addition` annotation are transformed so that they appear as ordinary classes only after a template instantiation has targeted them.

10.6 Related Work

A trait [31] is a construct that encloses a stateless[10] collection of provided and required methods. A trait may be used to compose new traits or as part of a class definition. The composition of traits is then said to be *flattened*. This entails that (1) the trait ordering in each composition is irrelevant, and (2) that a class composed from traits is semantically equal to a class in which all the methods are defined directly in the class. When used to compose a class, all requirements must be satisfied by the final composition. Traits were originally developed for the dynamic language Squeak, and support method aliasing and exclusion upon composition. A statically typed version also exists [26].

Mixins [4] are similar in scope to traits, in that they target the reuse of small units of code. Mixins also define provided and required functionality, and the main difference between them and traits is arguably the method of composition. Mixins traditionally rely on inheritance, by defining a subclass with as-of-yet undefined parent, and thereby requiring that mixins are linearly composed. In Groovy, however, mixins are mixed in at runtime using the `mixin` method of the `Class` class.

Functionality similar to that of traits and mixins can quite easily be mimicked with PT. For instance, to create a reusable collection of methods (with or without accompanying state), one might simply define a template with a single class, consisting of the methods that are subject to reuse. This class may then be merged with other classes where the functionality is needed. When it comes to specifying required methods, PT provides no such concept out-of-the-box, but a solution might be to define abstract and/or virtual methods in the template class, as we have shown for the Observer pattern example. As is the case with traits, merge/composition order is not significant in PT.

Perhaps the biggest conceptual difference between mixins/traits and PT comes in form of intended scope, in the sense that PT is targeted towards reusing larger units of code. In that regard, the former two can be seen as a special case of what can be accomplished with PT, admittedly with a slightly more involved syntax and some 'glue code'. In contrast to the Groovy mixin implementation (which makes global changes to existing classes at runtime), the PT approach is to define a new set of classes (that may be given a local scope) for each runtime instantiation of a template, so that such changes do not inadvertently affect code in unrelated parts of an application. One might therefore argue that dynamic PT offers a safer way to perform runtime class modifications, given that existing objects need not be affected.

Open classes in MultiJava [5] supports adding new methods to existing classes without resorting to subclassing or modifying the original code. (Multiple dispatch is also supported.) Several other languages, like Ruby, JavaScript and Groovy (and to a certain extent C#), also support open classes with varying syntax and capabilities. One of the main obstacles overcome in [5] has to do with static typing in the form of modular type checking and compilation, which is not an issue for us. With open

[10] Traits were originally defined to be stateless, although a more recent paper [3] has shown how a stateful variant may be designed and formalized.

classes, one can achieve some of the same end results as with PT, with some important distinctions. First, PT supports adding both state and behavior to existing classes, whereas with open classes only behavior can be added. Furthermore, PT allows the developer to make several different and unrelated additions and modifications, resulting in separate class hierarchies that may be localized to certain parts of an application. MultiJava does not support runtime addition of methods (probably just because it is tightly bound to Java), but the open classes in Ruby, JavaScript and Groovy support this.

Groovy also supports an open classes-like concept called *categories*, that enables the developer to specify that certain methods should be added to appropriate types inside a scope enclosed by curly brackets. The added functionality will apply to every statement inside this lexical scope, and also propagates down the call stack. However, only static methods are supported, and instance state cannot be added. As such, categories are similar to C# extension method [10].

BETA [23, 24], gbeta [9] and J& [27] (pronounced "jet") are systems that in many ways are similar to each other and in many respects can achieve similar end results to those of PT. A common property of all of them (except PT, that is) is that they utilize virtual classes (as introduced by BETA) to enable specialization and adaption of hierarchies of related classes. gbeta and J& support multiple inheritance, and this may to a certain extent be used to "merge" (in the PT sense of the word) independent classes. Neither BETA, gbeta nor J& support concepts similar to runtime template instantiations.

Aspect-oriented programming (AOP) [20] involves concepts related to PT. For instance, intertype declarations in AspectJ [6] may (statically) add new members to existing classes. The Caesar language [1, 25] supports both aspect-oriented programming constructs and code reuse and specialization through the use of virtual classes. It also supports wrappers for defining additional behavior for a class, and dynamic deployment of aspects at runtime (through use of the `deploy` keyword). In [33], Tanter describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects), however, these aspects may affect behavior only, and not class structure or hierarchy.

Context-oriented programming (COP) [7] provides a way to activate and deactivate *layers* of a class definition at runtime. Layer activation can be nested, and propagate down the call stack (for the current thread). This differs from PT, where runtime modifications to template classes will be visible according to the visibility of the variable that holds the instantiated package template (so for instance a `ThreadLocal` variable could yield different instantiations per thread). While doable, to mimic COPs stack-like activation and deactivation of features at runtime in PT, one would have to write quite a lot of 'glue code'.

In a subject-oriented [18] programming (SOP) system, different subjects may have differing views of the (shared) objects of an application. There is no global concept of a class; each subject defines 'partial classes' that model that subject's world view. What is called a *merge* in SOP, is somewhat different from a merge in PT. In SOP, a merge is an example of a composition strategy (and there may be many others), that tells the sys-

tem how to compose separate subjects with overlapping methods and/or state. Like with mixins and traits, there is a difference in intended scope when comparing SOP with PT; SOP targets a broader scope, with entire (possibly distributed) systems (that may even be written in different languages) being composed. One could, however, picture an extended PT-like mechanism as a basis for an implementation of SOP.

10.7 Concluding Remarks and Future Work

In this paper, we have described how package templates can be applied to and implemented in the dynamic language Groovy. We have argued, through a set of examples, that it provides powerful features for code modularization and runtime adaption of entire class hierarchies, and that this provides a set of advantages over conventional approaches to runtime class modifications.

For future work, it seems worthwhile to investigate the reconciliation of some of the dynamic features from this paper with the static version of PT, so that, for instance, templates may be instantiated at runtime in a type safe manner within a statically typed language like Java or C# .

Further utilization of the openness of this implementation, by providing the programmer with direct means to manipulate and add e.g. merge strategies and conflict resolution options, seems like an interesting endeavor that could add useful flexibility to our framework.

Applying the mechanism to a real-world scenario is also a priority in our continued research and development efforts in this area.

Acknowledgements

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30). The authors would like to thank the kind individuals on the *groovy-user* mailing list for helpful hints during the implementation process. Furthermore, we would like to thank Fredrik Sørensen and Birger Møller-Pedersen for fruitful discussions, and the anonymous reviewers for valuable feedback.

Bibliography

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] E. W. Axelsen, F. Sørensen, and S. Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 37–42, New York, NY, USA, 2009. ACM.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[6] A. Colyer. AspectJ. In *Aspect-Oriented Software Development*, pages 123–143. Addison-Wesley, 2005.

[7] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[8] Ecma International. *Standard ECMA-334 C# Language Specification*, 4th edition, 2006.

[9] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[10] Extension methods (C# Programming Guide). URL: `http://msdn.microsoft.com/en-us/library/bb383977.aspx`.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] A. Goldberg and D. Robson. *Smalltalk 80 : The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Professional, January 1989.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] The Grails homepage. URL: `http://grails.org`.

[15] The Groovy homepage. URL: `http://groovy.codehaus.org`.

[16] Groovy: Runtime vs compile time, static vs dynamic. URL: `http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic`.

[17] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173, 2002.

[18] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[19] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[20] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[21] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. To appear in the Journal of Object Technology (available now from `http://home.ifi.uio.no/steinkr/papers/`), 2009.

[22] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[23] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[24] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[25] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD '03*, pages 90–99, New York, 2003. ACM.

[26] O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[27] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, 2006. ACM.

[28] The Python homepage. URL: `http://python.org`.

[29] The Ruby homepage. URL: `http://ruby-lang.org`.

[30] The Ruby on Rails homepage. URL: `http://rubyonrails.org`.

[31] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[32] F. Sørensen and S. Krogdahl. Generic packages with expandable classes compared with similar approaches. In *NIK 2007*. Tapir akademisk forlag, 2007.

[33] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, 2008. ACM.

Chapter 11

Paper IV: Controlling Dynamic Module Composition Through an Open and Extensible Meta-Level API

Authors. Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller Pedersen.

Publication. Proceedings of the 6th Symposium on Dynamic Languages (DLS), 2010 SPLASH compilation.

Abstract. In addition to traditional object-oriented (OO) concepts such as inheritance and polymorphism, several modularization and composition mechanisms like e.g. traits, mixins and virtual classes have emerged. The Package Template mechanism is another attempt at providing a flexible mechanism for modularization, composition and adaption.

Dynamic languages have traditionally employed strong support for meta-programming, with hooks to control OO concepts such as method invocation and object construction, by utilizing meta-classes and meta-object protocols.

In this work, we attempt to bring a corresponding degree of meta-level control to composition primitives, with a concrete starting point in the package template mechanism as developed for the dynamic language Groovy. We wish to support a wide range of possible composition semantics, and to make such choices available to the developer through a meta-level API. This API should be extensible, and the semantic variations should be applied within well-defined scopes.

Categories and Subject Descriptors. D.3 **[Software]**: Programming Languages; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Classes and objects*; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Modules, packages*

General Terms. Languages, Design.

Keywords. OOP, Modularization, Dynamic Languages, Templates, Meta-Level.

11.1 Introduction

Having good tools for modularization, reuse and adaption of code is a recognized need in the domain of software development. However, it seems that the traditional object-oriented mechanisms often tend to fall short when it comes to dealing with high degrees of complexity paired with a desire for the right degree of modularity (for the problem at hand). Several approaches have been introduced to better cope with such issues, such as e.g. traits [27], mixins [5], and mechanisms based on virtual classes, e.g. [1, 4, 8, 21, 22, 24], to name a few.

Package templates (PT) have been introduced as another approach to this for statically typed languages like Java [11] in [18], and subsequently for dynamic languages like Groovy [12] in [2]. Through the use of package templates, a library developer may write entire packages in the form of (package) templates that, upon subsequent usages, may be adapted and customized to fit a variety of different application and/or business domains.

However, a common property of all of these mechanisms is that they support only a rather strictly fixed set of composition semantics, and that there are limited options for extending or modifying this set (short of changing the source code itself, which often is an undesirable option). For a given use case, the chosen semantics of a mechanism might be undesirable, regardless of the careful and thorough design that went into it. For instance, for traits the "composition order is irrelevant" [27, page 3], while for instance in the JavaFX's [9] implementation of mixins [7], a "mixin declared earlier in the extends list takes precedence over mixins declared later in the extends list" with respect to conflicting methods and fields; yet it is not clear that one variant is inherently better than the other for all usages. Obviously, the list of possible semantic variations in a language is virtually without bounds. Thus, *a mechanism that attempts to provide a greater degree of flexibility should in itself be extensible*.

For languages that support a meta-object protocol (MOP) [16], the application developer may control the semantics of method lookup and dispatch (beyond traditional virtual methods), field access etc. We propose that also for module import, adaption and composition, semantic control should be available to the programmer, and hence that the language's semantics should be open for modification through hooks or other fitting abstractions throughout the composition process. Such modifications to the semantics should have a well-defined, and optionally local, scope. Furthermore, through utilizing the MOP, many languages, including Groovy, allow objects to replace their own meta-object, and as such take control over their own semantics. Correspondingly, we propose that modules should be able to include code that utilizes meta-level functionality in order to programmatically control their own usage scenarios.

In this paper, we will focus on modules in terms of packages, and we propose a meta-level mechanism that allows the developer to interact with the runtime process of importing, adapting and composing such packages. Furthermore, the developer will be given the means to extend the mechanism in useful ways, such as e.g. adding new keywords with AOP-like functionality. Our approach is based on an extension of the dynamic package template mechanism for the Groovy language (which we shall refer

to as GPT) [2]. While the examples will be specific for Groovy and GPT, we believe that the mechanism and the problems discussed in themselves are, at least to a certain extent, general, and that it should be viable in and for a wide range of languages and composition mechanisms (though admittedly the language in question's support for reflection and the nature of the type system, be it static or dynamic, will be of great importance with regard to the ease of such an implementation).

An added motivation for this work is also to more fully grasp and reap the potential of the combination of dynamic languages like Groovy, and strong modularization mechanisms like PT; it seems that with the GPT basics in place, there is a wide range of possible and hopefully fruitful extensions that can be applied to further improve the expressiveness and flexibility of the mechanism, in keeping with tradition for such languages.

The contributions of this paper are: we present an extensible meta-level approach for controlling the semantics of software composition, and demonstrate the supposed validity of this approach on a few non-trivial examples. We also provide a proof-of-concept implementation in the form of a pluggable jar file for use in any Groovy program.[1] The main difference between our approach and other works on package/module composition is that we provide an open and extensible API that, in addition to requiring no extra artifacts in form of e.g. new compilers or composition rule languages, also opens up for a host of new ways to interact with the composition process.

The remainder of this article is structured as follows: In Section 11.2, we briefly describe the distinguishing features of the Groovy language, and the main concepts of PT for Groovy. (For a thorough treatment of the same concepts for PT for statically typed languages, the interested reader is referred to [18].) Section 11.3 describes the main concepts of our meta-level API. A set of motivating and detailing examples is presented inline with the different topics of this section. A brief discussion on the generality and applicability of this work can be found in Section 11.4. The implementation of the mechanism is briefly discussed in Section 11.5, and related work is discussed in Section 11.6. Section 11.7 concludes this article and touches on possible ideas for future work.

11.2 Background

11.2.1 Groovy

Groovy is a dynamic language that runs on the Java Virtual Machine. It can hence integrate nicely with Java, and to a certain extent it provides a familiar Java-esque syntax. The language rests on a purely object oriented foundation, and provides support for several attractive language concepts.

Both static and dynamic typing of variables and method signatures is supported in

[1] The prototype implementation can be downloaded from `http://home.ifi.uio.no/eyvinda/dls10`.

Groovy. However, there is presently no performance to be gained in Groovy by using static typing as opposed to dynamic (and in many cases it may actually be slower). Furthermore, Groovy cannot detect e.g. invalid calls to statically typed methods at compile time (since a fitting method may be added to the object in question at runtime) [13]. We will therefore not consider the static typing features of Groovy in this paper, and instead focus solely on dynamically typed references, methods and properties defined by using the keyword `def`.

A runtime meta-object protocol (MOP) is supported by the Groovy language. In accordance with the MOP, every object and class has an associated object of a meta-class. This meta-class object may be replaced by a version defined by the programmer, and this allows the semantics of method invocation and property getters and setters to be changed on a per-object or per-class basis. Also, new members may be added to classes and objects at runtime using the meta-class.

Further tailoring of the language's semantics may be done using compile-time transformations of the abstract syntax tree (AST). AST transformations may be defined by the developer to transform or replace parts of the syntax tree during different phases of the compilation process, such as parsing, semantic analysis, code generation etc. Such transformations may be global, or local pertaining to certain targets that typically may be Java-like annotations on source code elements. Dynamic typing aside, one thing in Groovy that is resolved entirely at compile-time is the given class from which to create a new object (e.g. as in `new A()`). That is, if a referenced class `A` cannot be statically determined to exist within imported packages and with correct accessibility modifiers, a compile-time error will occur. Since we will be loading packages dynamically, we will use compile-time AST transformations to get around this issue.

Closures are first class citizens of Groovy, and units of code may as such be passed to and from methods. Closures are hence also objects, and have their own meta-class object that may be replaced by the programmer. It is worth noting, however, that Groovy closures are not actually closures in the usual sense of the word, since they do not necessarily *close* around the variables to which code within the closure refers (i.e., closures in Groovy can contain unbound free variables). We shall nevertheless in this paper use the term *closure* to refer to the syntactic constructs that define subclasses of `groovy.lang.Closure`, and instances thereof.

Closures might have a *delegate*, to which method calls and field accesses can be dispatched (by setting the closure's resolution strategy to for instance `Closure.DELEGATE_FIRST`). The delegate is, as the meta-class object, replaceable by the developer.

A closure is written within curly braces, and may take parameters (or be parameterless). Consider a small example adapted from [12]:

```
def c = 1
def printSum = { a, b -> print a + b + c }
printSum(5, 7) // prints 13
```

Now, by setting `printSum.delegate` to an object that contains a field named `c` with a value that is different from 1, and setting the resolution strategy as mentioned above,

we can change the printed output, and this, together with MOP features, provides for flexibility that proves to be very useful when implementing an open and extensible API such as the one we are about to describe.

Parentheses and semicolons in method calls can be omitted in Groovy. Thus, for a method `m` that takes a closure as its sole argument, the call can be written as `m { ... }`, where the ellipsis represents the actual code of the closure.

11.2.2 Groovy Package Templates at a Glance

As mentioned in the introduction, the package template mechanism was introduced in [18]. In that paper, Java was the target language for the mechanism, and a detailed exposition of the features of the original version of the mechanism can be found in that paper. We later adapted the main concepts of the mechanism to dynamic languages, with particular focus on Groovy in [2], and it is the latter version that will be the basis for this paper. Below we explain the basic concepts of the mechanism (most of which indeed are common for both static and dynamic versions), before we move to the meta-level functionality in the following sections.

The package template mechanism targets the development of reusable and adaptable collections of interdependent classes. A package template looks much like a regular Groovy file, with the most notable exception being a `template_def` construct enclosing classes and interfaces. An example is shown below:

```
template_def T {
   class A { ... }
   class B extends A { ... }
}
```

The contents of a template are ordinary Groovy classes or interfaces. These types may refer to each other and take part in type hierarchies within the template. A template class is not allowed to have inheritance relationships to classes defined in packages outside the template (with the exception of classes in package templates that are instantiated within the current template, more on instantiations below). Templates need to be semantically (and, of course, syntactically) valid in their own right, and may as such be checked for validity according to the rules of the target language (in this case Groovy) independently of their potential usage.

None of the classes or interfaces defined in a template are available to the rest of the program before the template itself is instantiated (that is, the class loader does not see any of the members of a template before an instantiation of the template has been performed).

A package template must hence be explicitly instantiated before any of its contents can be used. A template can be instantiated multiple times, and each instantiation of a given template is completely independent of any other instantiations. The result of an instantiation is a new package (we shall, in this paper, use the term *package* in a broad sense, meaning a collection of classes and interfaces that is internally consistent, but not necessarily implemented as a regular Java or Groovy package as such). A

package template is instantiated at runtime by utilizing the `instantiate` method of the framework's `PT` class, as shown in the following construct:

```
def INST_T = PT.instantiate {
   template T {
      map B, BB
} }
```

This statement[2] will make *a copy of* all the classes and interfaces defined in the template `T` available to the caller of the static `instantiate` method through the variable `INST_T`[3], preserving potential type hierarchies as defined in the template. Classes within the new package can be accessed by qualifying their name with the variable name, e.g. as in `new INST_T.A()`.

The sub-statement "`map B, BB`" entails that the class `B` (and all explicit references to that type) should be renamed (retyped) to `BB`.

However, instantiations may also be more elaborate: the user can specify that classes or methods should be renamed, that classes should have additions, and/or that (previously unrelated) classes should be merged.

To facilitate this, implementation-wise, the `PT.instantiate` method takes a Groovy closure as its sole argument, and this closure may in turn contain several calls to the `template` method. The `template` method takes a template name (such as `T` in the example above), and optionally a specification (in the form of a closure) of *adaptations*, such as mappings to new names etc. Thus, the instantiation specifications provided to the `instantiate` method can be viewed as expressions in a tiny internal DSL for instantiating package templates.

If multiple templates are listed in the argument provided to `instantiate`, this will lead to *one single* instantiation of these templates, forming *one single* new package.

Consider now a second, equally simple, template U:

```
template_def U {
   class C { ... }
}
```

Taking the example templates `T` and `U` from above, we can employ the following call to merge the classes `A` and `C` to form one new class `AC`:

[2] As mentioned in Section 11.2.1, the syntax of Groovy is quite flexible, allowing the developer to omit parenthesis and semicolons where desirable. Furthermore, the GPT framework will manipulate the resulting expression a little at runtime, to allow the developer to omit commas and quotation marks. The `instantiate` statement could hence also be written in a more elaborate (and perhaps more immediately familiar) way: `def INST_T = PT.instantiate({ template("T", { map("B", "BB")}) });`.

[3] The variable name holds no significance, but as a convention we will name variables holding instantiated templates `INST_` followed by the template name(s) or a derivation thereof.

```
def INST_TU = PT.instantiate {
    template T { map A, AC }
    template U { map C,  AC }
  }
```

Here, the classes `A` and `C` are both renamed (or *mapped*) to `AC` (using the `map` method), and they will thereby be merged according to the semantics of the mechanism (which, roughly, comes down to taking the discriminated union of all the attributes from each class involved in the merge). A new, identical (from a textual point of view), copy of the class `B` will be available through `INST_TU`, while the template classes `A` and `C` will not be available as individual classes, but only as their merge. The class `B` will now be a subclass of the new class `AC`, and all existing references to `A` within `T` and `C` within `U` will be retyped to `AC`. Already at this point, we clearly see that there are several possible options for semantic variations in the language, with respect to merge order, conflict resolution semantics etc.

PT also allows for additions to be made to instantiated classes, so that methods may be overridden and new methods and fields may be provided. In GPT this is done by writing an ordinary class with the same name as the one to which additions should be made, and decorating it with a `@PTAddition` annotation. The concept of additions is important in the (G)PT mechanism, but we will not make any further use of it in this paper, and hence we will not discuss it in any further detail. (The interested reader is referred to [2, 18] for a more detailed exposition.)

11.3 The Meta-Level Package Instantiation API

In this section, the main body of our work will be presented. We will start with a general overview of the main concepts of the mechanism and its application and scope in Sections 11.3.1 through 11.3.3. Then, in Section 11.3.4, we will look at how this can be applied to provide different options for resolution of potential conflicts in the composition process, by utilizing the possibilities of the meta-level API. As promised by the title of this paper, the API itself is also *extensible*, and in Section 11.3.5 we will explain how this can be utilized to provide convenient access to new functionality, exemplified through an (admittedly simple) AOP-like construct. Another main idea of this work is that units of code, such as package templates, can contain introspective meta-level code in order to control and customize their usage; this will be described in Section 11.3.6. Interactions between different meta-level pieces of code are briefly treated in Section 11.3.7.

11.3.1 Overview

When a module/package is utilized within a program, there needs to be a specification of how the module should be made available. Such specifications may have varying degrees of expressiveness and variation points. For e.g. a regular package in the Java language, a rather simple specification in form of the `import` statement is used.

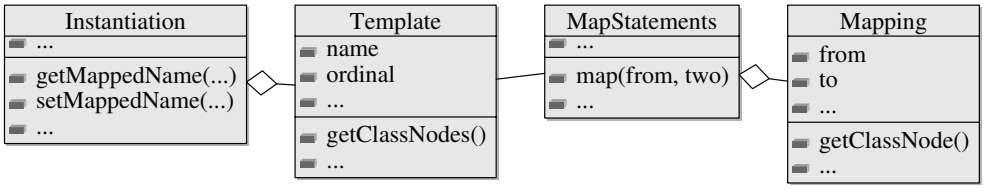

Figure 11.1: A slightly simplified view of the classes representing an instantiation of package templates.

Common use cases are to specify that all the classes in a given package should be imported, e.g. `import javax.swing.*`, or that a specific class should be imported, e.g. `import javax.swing.JOptionPane`. The modifier `static` may also be applied to allow static class fields and methods to be referenced without qualification.

For both the static and dynamic versions of PT, the developer can write specifications that describe the instantiation of package templates to form packages that are subsequently imported. These specifications can specify a significantly wider range of options compared to their Java counterpart (e.g. additions, merging, renaming etc), but the set of options is still fixed.

In the following, we will present a meta-level API for interacting with the instantiation and composition of templates and their contained classes. This API can be seen as a meta-level instance of the *strategy design pattern* [10], in which the provided semantics of the GPT mechanism as described in [2] can be seen as a base case, and different strategies may be supplied for different semantics. An alternate view is to look at this from an *open implementation* [15] point of view, in which the methods of the GPT framework from [2] would be the primary interface, and the API presented here would be the meta-interface. The functionality exposed through the primary interface would then be open for semantic customization through the meta-interface. Thus, the primary interface for a given package abstraction will to a certain extent dictate what is reasonable to include in the meta-interface.

In our concrete instance with GPT, the API will be closely tied to runtime calls to the `instantiate` method of the `PT` class. When the `PT.instantiate` method is called, as exemplified in the Section 11.2.2, a tree-like object structure representing the desired instantiation, as specified by the closure supplied as the actual parameter of the call, is created. A simplified view of the classes involved in this tree is shown in Figure 11.1. An object of the `Instantiation` class will be the root of the tree; it contains a list of templates to be instantiated, represented by objects of the `Template` class. `Template` objects contain a collection of `Mapping`s, that are held by a reference to an object of the `MapStatements` class. This representation of an instantiation is an important part of the API, as it will be passed on to most of the methods invoked at the meta-level (see next section for details). This enables the developer to examine and manipulate the instantiation specification in order to control the semantics that are applied when

creating the resulting package.

11.3.2 Meta-Level Interception Points and Strategies

In the same way that meta-object protocols typically provide *hooks* into e.g. method dispatch and object creation, our mechanism needs to provide sensible hooks into the package template instantiation and composition process. We will refer to such hooks as *interception points*. An obvious such point at which interception seems fruitful is when the object tree representing the instantiation specification (as objects of the classes in the class diagram in Figure 11.1) has been created. At this point, the specification (including which templates to instantiate, which classes to merge, etc.) might be changed by meta-level code.

The GPT framework provides access to the meta-level API by allowing the users to define classes that implement one of a set of interfaces provided by our framework. One such interface is the `PTInstSpecStrategy`, shown below:

```
interface PTInstSpecStrategy {
   // The following method will be called once,
   // when a representation of an instantiation
   // specification has been created:
   def processInstantiationSpec(instantiation);
}
```

We shall refer to implementations of interfaces such as the one above as *strategies*. The flowchart in Figure 11.2 shows a conceptual view of the template instantiation process performed by our framework. First, the framework checks for so-called *setup strategies* (`A`). After the potential application of any such strategies, a representation of the instantiation specification is built. Then any strategies implementing `PTInstSpec-Strategy` are applied, at the interception point marked `B` in the figure. (We shall return to point `A` in greater detail later on, in Section 11.3.5.)

Subsequently, an abstract syntax tree is built for each template specified by the instantiation. At this point, the instantiation specification has been processed (and is thus considered complete). The framework then checks for supplied implementations of the `PTPreInstStrategy` interface, shown in the code snippet below, to allow changes to the template ASTs before the specified (and possibly modified) instantiation is carried out.

```
interface PTPreInstStrategy {

   // called for each template class:
   def processClass(classNode, template, instantiation);

   // called for each template interface:
   def processInterface(interfaceNode, template,
      instantiation);
```

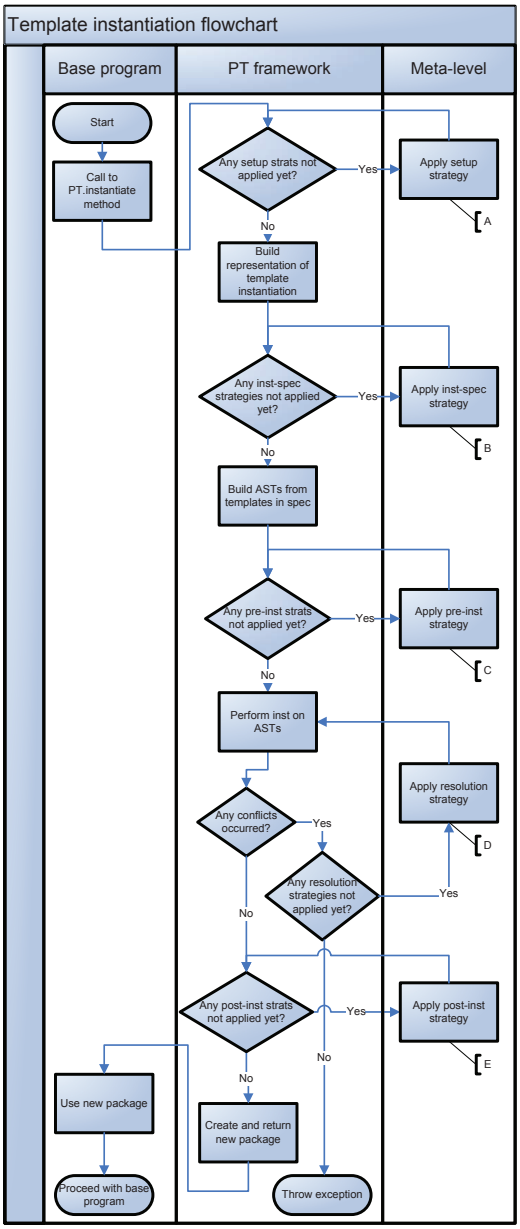

Figure 11.2: A flowchart showing a conceptual view of the template instantiation process. The available meta-level interception points are labeled from `A` to `E` in the right-hand column in the chart. For each interception point, strategies are (by default) applied in the order they are listed, see Section 11.3.3 for details.

```
   // called for each specified class merge:
   def processMerge(classes, instantiation);
}
```

Implementing strategies according to this interface allows for modification of classes (e.g. adding, renaming or removing methods) before the instantiation is carried out on the AST (i.e. renames, merges etc has not yet taken effect). A corresponding interface `PTPostInstStrategy` is also available, allowing for modifications to the AST after the instantiation specification has been performed, as shown below:

```
interface PTPostInstStrategy {

   // called for each instantiated class
   // (resulting from a merge or not):
   def processClass(classNode, instantiation);

   // called for each instantiated interface
   // (resulting from a merge or not):
   def processInterface(interfaceNode, instantiation);
}
```

Note that at the time that implementations of `PTPostInstStrategy` are invoked, any specified merges have already been performed on the AST, and the interface thus has no `processMerge` method, as opposed to the pre-instantiation version.

In between the pre and post instantiation steps, the actual instantiation is carried out on the AST. During this process, a set of conflicts that prevent the instantiation from being completed successfully might arise. Situations such as this can be handled at the meta-level at interception point `D` in Figure 11.2, with implementations of the `PTConflictResolutionStrategy` interface, as shown below:

```
interface PTConflictResolutionStrategy {

   // invoked upon each method conflict
   // resulting from a merge:
   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation);

   // invoked upon each constructor conflict
   // resulting from a merge:
   def onConstructorConflict(classNode1, constrNode1,
      classNode2, constrNode2, instantiation);

   // invoked upon each field conflict
   // resulting from a merge:
```

```
    def onFieldConflict(classNode, fieldNode1,
       classNode2, fieldNode2, instantiation);

    // invoked for merges that would lead to
    // multiple inheritance
    def onSuperclassConflict(classNode1, classNode2);

    // corresponding signatures for conflicts
    // within merged interfaces here...
}
```

By creating classes implementing `PTConflictResolutionStrategy`, the developer is free to design alternate conflict resolution semantics; we shall take a closer look at this in Section 11.3.4.

As we have seen, the methods of the interfaces presented in this section take formal parameters named `classNode`, `methodNode`, `fieldNode` etc. The GPT framework will supply the actual parameter values at runtime, by creating objects of classes named `PTClassNode`, `PTMethodNode`, `PTFieldNode` etc. (`PTClassNode` is also the default class for the objects returned from the `getClassNode(s)` methods of the `Template` and `Mapping` classes from Figure 11.1.) These classes contain convenience methods with which the developer can interact in order to modify the AST of template classes and interfaces, as will be illustrated in the examples throughout the rest of Section 11.3.

The `PTClassNode` class contains the following methods, for which a sufficient understanding of their semantics should be inferable by their name: `getName`, `setName`, `getMethods` (which will return a collection of objects of the `PTMethodNode` class with standard collection operations for adding and removing objects), `getFields`, `getConstructors`, `getSuperClass`, `getInterfaces` and a protected `getAST`. The `PTMethodNode` class contains methods `getName`, `setName`, `getSignature`, `setCode` and a protected `getAST`. The `setCode` method probably deserves a comment; it takes a Groovy closure as its single formal parameter, and uses that for specifying both the signature of the method and the actual code of the operation. The method will be injected in the new class through the use of Groovy's meta-object protocol. An example usage of `setCode` will be shown in Section 11.3.4. For brevity we will skip listing the operations on the `PTFieldNode` and `PTConstructorNode` classes. However, it is worth noting that through the extension mechanisms described in Section 11.3.5, and by utilizing the `getAST` methods, one can extend the set of available operations for each of these classes.

11.3.3 Strategy Application and Scope

There are at least two valid scopes in which a strategy could apply, and that is global, i.e. for every instantiation throughout a program, and local, pertaining only to one specific instantiation. For the global scope, the `PT` class provides an `addStrategy`

method, that may be called to add a strategy which will remain in effect for every subsequent call to the `instantiate` method (unless a corresponding call to the `removeStrategy` method has been made).

The other option is to supply the strategy directly as part of the instantiation statement, by using the `strategy` pseudo-keyword. In the example below, `MyStrat` will be applied to the instantiation of `T` and `U`, and will then be discarded (and subsequently garbage collected):

```
PT.instantiate {
   strategy MyStrat {
      template T
      template U
} }
```

A slight variation of the above is to specify a strategy only for a part of an instantiation. In the example below, the strategy will only apply to the `U` template. Thus, anything that involves `T`, such as a merge of classes from these two templates, will not be affected by the strategy.

```
PT.instantiate {
  template T
  strategy MyOtherStrat {
     template U
} }
```

Specifying a strategy in this way also has the added implication that only interception points `A`, `B` and `C` from Figure 11.2 are valid for strategies that do not cover an entire instantiation. This is due to the fact that any later interception points will potentially involve data from all templates specified in the instantiation (and the strategy should thus have been at the outermost level if access to such data is required).

If more than one strategy is listed in the call to the `instantiate` method, they are by default ordered according to the following rules: first, each interface pertains to a specific interception point, and the order in which these points are encountered is, as shown in Figure 11.2, from `A` to `E`. Secondly, strategies are executed in the same order as they are listed in the instantiation specification, depth first.

For local strategies, there is also the issue of *propagation* (which would be an issue for instance when code in a template instantiates other templates). With focus on aspect deployment, Tanter has treated the issue of deployment scope, and identifies three dimensions: propagation down the call stack, propagation with regard to delayed evaluation (execution of methods on objects created within a scope, after the scope has been exited) and deployment-specific join-point filtering [29]. While the latter has little relevance to us (since join-points are specific to AOP), the first two are clearly relevant. In this work, we settle for the base case of no propagation, neither down the call stack nor to delayed evaluations, and explicitly leave investigation of the other variants for future work. This means that templates instantiated within templates in an instantiation in which a local strategy is in effect, will not be affected by said strategy. That is,

if a given template `A` (or a class within it) instantiates a template `B`, and template `A` is instantiated with a local strategy `S`, `S` will *not* apply to the instantiation of `B`. However, a local scope that propagates down the call stack can be emulated by adding a global strategy at the first interception point, and subsequently removing it again at the last.

By utilizing the different options for specifying strategies, it is possible to both apply different composition semantics to a controlled portion of the code, and also to mix different semantics in one program, by utilizing different strategies. Furthermore, one can base strategy selection upon runtime conditions (e.g. such as whether a debug or trace flag has been set).

11.3.4 Alternate Semantics for Conflict Resolution

When performing a merge of previously independent (template) classes to form a new class, several conflicts may occur. The basic strategy employed by GPT (and, indeed, the static version of PT) is to require an explicit resolution from the programmer. However, it is not at all obvious that any given strategy is preferable for every case. For instance, when a conflict due to duplicate method definitions occur, there are at least a few reasonable approaches to take. The framework could view methods belonging to the class listed first in the call to `instantiate` as having precedence, and discard any conflicting methods stemming from classes appearing later in the instantiation statement (this, i.e. using order to resolve conflicts, is, for instance the approach taken when resolving members originating from mixin inheritance in the JavaFX language [9]). Another approach would be to somehow compose the two (or more) methods, so that the corresponding method in the new class would yield the effect of calling one of the original methods after the other. The latter approach (or variants thereof) could be relevant if trying to achieve similar effects (albeit on much a smaller scale) as is attainable with composition mechanisms such as those available in e.g. HyperJ [26].

To override GPT's default method conflict resolution strategy (which is to throw an exception unless an explicit resolve is present), and provide a strategy similar to that of JavaFX mixins, we could employ the following implementation of `PTConflict-ResolutionStrategy`:

```
class OrderSignificantMergeStrat implements
   PTConflictResolutionStrategy {

   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation) {

      // remove the method from the class from the
      // last of the two corresponding templates
      classNode2.getMethods().remove(methodNode2)
   }
```

```
   // corresponding implementation for field conflicts
   // here, and (possibly) empty implementations of
   // the remaining methods of PTConflictResolution-
   // Strategy
}
```

To utilize this particular strategy when performing an instantiation, we could for instance supply it explicitly to the `PT.instantiate` method by utilizing a `strategy` pseudo-keyword, or utilize the static `addStrategy` method of the `PT` class, as discussed in Section 11.3.3. An example of the former approach is shown below:

```
PT.instantiate {
   strategy OrderSignificantMergeStrat {
      template U
      template T
} }
```

Given the instantiation above, any conflicting method signatures in merged classes would be resolved by preferring those stemming from the template `U` over those stemming from `T`. Clearly, variants of this strategy (e.g. renaming of conflicting methods) could easily be created by making slight modifications to the strategy above.

Yet another approach would be to disregard method conflicts entirely. Given a situation in which only a (well-known) subset of the templates' functionality is of interest to the user, it might be reasonable to disregard conflicts for methods that will never be called at runtime anyway. Providing the following strategy would enable such an approach:

```
class IgnoreConflictsStrat implements
   PTConflictResolutionStrategy {

   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation) {
      // remove one of the methods so that the code
      // will compile:
      classNode.getMethods().remove(methodNode2)
      // replace the code of the other method
      // with just an exception:
      methodNode1.setCode(
         { () -> throw new Exception(...) })
   }
   // implementation of the rest here...
}
```

Such an approach as the one above clearly requires good unit test coverage in order to prevent exceptions from popping up at unexpected places at runtime. However, given

such coverage (which doesn't seem too far-fetched in the context of a system developed in a dynamic language to begin with) it could simplify merging and template usage in general.

11.3.5 Extending the Instantiation DSL

At interception point `A` in Figure 11.2, there is a check for *setup strategies*. Such strategies may be applied in order to customize the way an instantiation tree representation is built from a specification, or to customize other parts of the GPT framework. We shall, as an example, look at how we can utilize this in order to provide a richer DSL for setting up instantiation specifications. The GPT framework will at interception point `A` look for classes that implement the `PTSetupStrategy` interface:

```
interface PTSetupStrategy {
   // called once upon initialization of the
   // instantiation process:
   def initialize();

   // called once to acquire the factory to
   // use when creating objects:
   def getFactory();
   ...
}
```

The `getFactory` method above utilizes the *abstract factory* [10] design pattern in order to return a factory; this factory implements an interface named `PTClassFactory` for creating objects of the classes used by the `PT` class of the GPT framework. The framework contains a default implementation (that can return objects of the classes from Figure 11.1), but through the use of strategies, the user might change which factory will be used, and thus which classes are used for nearly every part of the instantiation process.[4]

A Simple AOP-like Extension

As an example extension, we shall consider adding simple AOP-like functionality that allows for specifying a set of classes with which functionality from one class should be merged. We shall utilize this to implement the typical AOP idiom of adding logging before or after method invocations to every class in a given subset of a program; here this subset will be some of the classes from a template. One option would be to specify each class in this subset explicitly, however, for non-trivial templates this would be both tedious and error prone. Thus, a better option would be if the framework could support more complex expressions for selecting classes for merge in the instantiation

[4] And thanks to the dynamic nature of Groovy, the new classes of which the factory returns objects need not have any relation to the existing ones other than in their ability to answer to the same method calls, further substantiating the flexibility of this approach.

specification, e.g. that classes having names containing a given string or classes containing certain method signatures should be included. We will shortly get back to how we can express that, but first we will look at implementing the logging functionality.

For logging method calls, we will utilize a simple template `Logging`, with a single class named `Logger`. This class will use the Groovy MOP (utilizing the `GroovyInterceptable` interface) to intercept each method call, and then log the invocation:

```
template_def Logging {
   class Logger implements GroovyInterceptable {
      def invokeMethod(name, args) {
         this.&log(name, args) // log the call
         metaClass.invokeMethod(name, args) // proceed
      }
      // implementation of log method here...
} }
```

(The ampersand in the call to the `log` method bypasses the MOP, in order to prevent infinite recursion.)

In order to use the `Logging` template to log method invocations for several classes, one possible syntax for this extension could be as follows:

```
PT.instantiate {
   strategy PTMapExtensionsStrategy {
      template ToBeLogged
      template Logging {
         mapOntoEach Logger, ToBeLogged,
           { c -> c.getName().contains("...") }
} } }
```

The `mapOntoEach` method shown above does not exist in the framework yet, but the desired semantics are as follows: the first parameter is the name of the class that we want to map to each of the classes as specified by the second and third parameter. The second parameter is the name of the template from which to get these classes, and the third parameter is a predicate in the form a closure that takes a ClassNode (from Figure 11.1) and returns a bool. The method should create a new mapping in the specification from, in the example above, `Logger` to every class in the `ToBeLogged` template that satisfies the predicate (which in the concrete example above is that the name of the class should contain a specified string, represented here by an ellipsis).

In order to implement support for this functionality, we may employ the following implementation of `PTSetupStrategy`, in which we replace the default class factory with our own custom version:

```
class PTMapExtensionsStrategy implements PTSetupStrategy {
   def setupFactory() {
      return new ExtensionsFactory()
   }
  // implement rest of the methods of PTSetupStrategy here...
}
```

The `ExtensionsFactory` class is a trivial implementation of the `PTClassFactory`, where the only significant method (for this use case) is the implementation of the `createMapStatementsNode`, which should return a subclass of the `MapStatements` class shown in Figure 11.1:

```
class ExtensionsFactory implements PTClassFactory {
   def createMapStatementsNode() {
      return new MapAllExtension()
   }
   // rest of the PTClassFactory methods here...
}
```

The `MapAllExtension` class shown in the following contains the gist of the new functionality, by defining the `mapOntoEach` method that is used in the instantiation above. The execution of the call in the instantiation specification is delayed until the closure in question is run, and method calls are then delegated to the `MapAllExtension` class.

Implementing the required functionality can be done with relative ease, by utilizing the fact that a `MapStatements` object has access to the instantiation, and from there one can easily get hold of the class names in a given template, as indicated by the `getClasses` method of Figure 11.1. The template is accessed through a collection (`templates`) on the instantiation that is indexed by the templates' name, and the `map` method utilized within the `each` loop[5] to perform the actual mappings is directly available from the `MapStatements` class.

```
class MapAllExtension extends MapStatements {
   def mapOntoEach(String classToMap,
         String templateName, Closure predicate) {

      instantiation.templates[templateName].
         getClasses().each {
           if(predicate(it)) {
              map(classToMap, it.getName())
} } } }
```

Clearly, more elaborate schemes for selecting classes than just enumerating every single one within a template an applying a boolean condition can be created. Still, this

[5]The variable `it` utilized within the loop is supplied by Groovy as an automatic reference to current object from the collection over which the loop is being performed. Thus, in the example `it` will be a reference to an object of the `ClassNode` class.

simple example, in our opinion, demonstrates one important aspect of the flexibility of our framework, i.e. the ability to easily extend not just the semantics of existing functionality, but also to add whole new concepts to the mix.

11.3.6 Meta-Aware Package Templates

The example strategies presented above have all been specified as part of an instantiation specification in the class that calls the `PT.instantiate` method. However, part of the motivation behind the original package template mechanism, and indeed this work as well, has been to facilitate the creation of self-contained, reusable entities that typically span multiple classes. From that point of view, there is another interesting possibility: to take advantage of the fact that all the power of our meta-level API can be made available to the package templates themselves, not only when they instantiate other package templates, but also as a way to hook onto their own instantiation process. This entails that templates in GPT may be introspective with respect to their own instantiation, and e.g. specify instantiation behavior, constraints or requirements that are inherent to the templates themselves, relating for instance to the target classes of a merge.

To utilize a strategy within a template, the template needs to contain an implementation of one or more of the strategy interfaces defined in the text preceding this section. In addition to implementing the interface, an annotation `@PTMeta` is required, to enable the GPT framework to cleanly separate meta-level template classes from base-level ones.

Meta-level template classes may as such not be explicitly referred to as part of an instantiation specification, and will thus not be part of the resulting package. Strategies internal to a template are at instantiation time applied prior to strategies (implementing the same interface) specified explicitly in the call to `instantiate`.

A Reusable Implementation of the Active Object Pattern

We have in previous work [3] discussed the application of package templates to provide reusable and pluggable implementations of some design patterns from [10]. Here, we consider how this can be done for another pattern that to begin with does not lend itself very well to pluggability, by taking advantage of meta-level strategies within templates.

The *Active Object* pattern is a design pattern for concurrent applications, and is described in detail in [19]. The pattern revolves around four roles: the *servant* or *active object* performs operations at the request of *clients*. A proxy sits between these two, and routes requests (i.e. method calls) through a *scheduler*, which in turn schedules and, subsequently, starts operations on the servant. The proxy immediately returns a *future* [20] to the client in place of the actual return value of the operation. The future contains methods for checking for the completion of the operation, and a (potentially) blocking method to wait until the result is ready.

Given the inherent relative complexity of this pattern, having a reusable, pluggable,

implementation would supposedly be a major convenience. Below we show a skeleton of the involved classes as a package template, with pseudo-code enclosed in angled brackets:

```
template_def ActiveObjectPattern {
  class Scheduler {
    def queue = < a fitting queue abstraction >
    synchronized def enqueue(operationName, args) {
      def future = new Future()
      queue.put(operationName, args, future)
      < start scheduler thread if not started >
      return future
    }
    < actual scheduling code here >
  }

  class Future {
    private def value
    private def completed = false
    synchronized def getValue() { ... }
    synchronized def setValue(val) { ... }
    def isCompleted() { return completed }
  }

  class ActiveObject { /* see below */ }
  class ActiveObjectProxy { /* see below */ }
}
```

For optimal ease of use and conceptual simplicity for the developer, it would be desirable if one could just merge the `ActiveObject` class with an appropriate domain class, and then proceed to use the latter regardless of its newly added active behavior. The syntax that we would like to have is something resembling the following instantiation and subsequent usage:

```
def INST_H = PT.instantiate {
   template HeavyLifting // contains ComplexProcess
   template ActiveObjectPattern {
      map ActiveObject, ComplexProcess
} }

def c = new INST_H.ComplexProcess()
def result = c.PerformLongRunningTask()
// do something else for a while...
// then (potentially) block and wait for the result
println result.getValue()
```

In order to achieve this, we would need a transparent proxy to handle the wrapping of method calls. Luckily, such proxies are relatively easy to create in most languages with proper support for a MOP, including Groovy. Hence, we may implement the `ActiveObjectProxy` as follows:

```
class ActiveObjectProxy implements
      GroovyInterceptable {
   static def scheduler = new Scheduler()
   def invokeMethod(name, args) {
      return scheduler.enqueue(name, args)
} }
```

This proxy simply intercepts every method call (as implied by the implementation of the `GroovyInterceptable` interface), and translates each call into a request that is enqueued on the scheduler. The scheduler will, as previously shown above, immediately return a future.

The active object itself, represented by the `ActiveObject` class in the template, is really only a placeholder for the functionality provided by the domain class. As such, we may actually leave this class completely empty in the template definition. However, since we are interested in *substituting* (from the user's point of view) the domain class with the proxy, and then subsequently routing calls to it through the scheduler, we need to employ a strategy to handle this. Since this strategy is a part of the pattern implementation itself, it makes sense to keep it within the `ActiveObjectPattern` template. The strategy is actually quite simple, and its only goal will be to remap the active object and proxy template classes, respectively.

```
template_def ActiveObjectPattern {
   @PTMeta
   class MappingStrategy implements PTInstSpecStrategy {
      def processInstantiationSpec(inst) {
         def aoName = "ActiveObject"
         def proxyName = "ActibeObjectProxy"
         def implName = "ActiveObjectImpl"

         inst.setMappedName(aoName, inst.getMappedName(aoName))
         inst.setMappedName(inst.getMappedName(aoName), implName)
         inst.setMappedName(aoName, implName)
   } }
   < rest of the pattern classes here >
}
```

By supplying the strategy above along with the classes representing the actual roles of the template, we allow the template to capture and encapsulate the conceptual intent of the pattern, which is to substitute a synchronous long-running task with an asynchronous approach through queueing requests. The user may express his or her intent

by mapping the active object to an appropriate domain class (representing the time-consuming task), and not worry about how the proxy works (or even caring that there indeed is one).

11.3.7 Interactions

Strategies might (obviously) interact with one another, and this might be deemed unfortunate. For instance, a strategy specified as part of an instantiation specification might interact with a strategy internal to the template(s) being instantiated, without the developer of the former even being aware of the latter (thus, what might be considered an advantage from an encapsulation and ease-of-use point of view might be seen as a hinderance or inconvenience in situations with multiple strategies).

Since strategies are developed in imperative code (and that imperative code might be highly dynamic and type-less), there is no good way to automatically determine interactions up-front. Thus, the strategy developer needs to take special care to document the workings and assumptions of the strategy, and the subsequent user needs to order strategies according to the desired results. If only one strategy (or a specific set of strategies) is required, an `exclusive` modifier may be applied to the `strategy` keyword in an instantiation, e.g. as shown below:

```
PT.instantiate {
  exclusive { strategy S {
     template T
  } } }
```

This would prevent any other strategies than `S` from executing.[6] However, the `exclusive` clause is merely a small stroke on a large canvas of ways to deal with interactions, and this is clearly an area in which more research could be fruitful.

11.4 Discussion

Even though the meta-level functionality presented in this paper is built on top of the basic GPT mechanism as presented in [2], which is an implementation of the package template mechanism specifically targeted at the Groovy language, it is our opinion that the mechanism has a great deal of conceptual generality, making it applicable to other composition mechanisms and other programming languages besides Groovy.

We observe that a composition in general relies on a specification of what is to be composed and how, and for GPT this corresponds to the instantiation specification. Other examples include Groovy's own runtime mixin mechanism, for which the specification is the classes mentioned as arguments to the `mixin` method, and the Traits

[6] The `exclusive` pseudo-keyword could with relative ease be implemented by a global strategy in the manner described in Section 11.3.5 (carefully making sure of not including itself in the exclusivity, obviously).

mechanism [27], for which the specification would be the set of traits and potential renames and exclusions specified in the trait list of a class.

Furthermore, we suggest that a representation of such composition specifications can be made available for modification by implementations of meta-level interfaces that are tailored to the composition mechanism at hand, and that such implementations can subsequently be invoked throughout the runtime composition process in order to control it. Applying this to e.g. the Traits mechanism, the meta-level interfaces could offer methods to be called for trait composition, for method exclusions and for conflict resolution, in much the same way that our framework does for template class composition. The pre and post instantiation/trait composition interception points could also still apply.

However, the overall applicability of our approach is hard to properly assess, both in terms of generality with regard to different composition mechanisms and in terms of usability for the programmer, without having applied it to a larger set of real-world scenarios. Thus this, necessarily paired with lifting the implementation from a proof-of-concept level, are important directions for future work. Nevertheless, the examples presented herein can at least serve to give us an idea of the potential usability of this work, and as such motivate a more thorough investigation.

11.5 Implementation

As mentioned earlier, the meta-level mechanisms of this work are built on top of work that we described in [2]. In that work, the basic GPT mechanism (or, in the terminology of [15], the "primary interface"), without the meta-level mechanisms presented here, was developed. By keeping, to a large extent, the previous implementation of basic GPT, many of the design choices made back then also applied this time around, e.g. to only make use of the standard features of Groovy itself, and not modify any of its source code, and to implement the functionality as a Java archive (jar) that can be included in any Groovy program in order to utilize GPT. One of the things that we touched on in [2] in that respect was the trade-off between a clean syntax (like the one in standard PT [18], that requires a dedicated parser), and ease of implementation by utilizing a tiny internal DSL to express instantiations. With this work, it seems clear that keeping everything in standard Groovy has paid off, since the addition of meta-level features could be done in a relatively straightforward and open manner. However, were it only for the ease of implementation it would perhaps not be all that significant, but the choice to keep the entire mechanism within the realms of Groovy means that it also enables the user to extend and modify the framework with relative ease. In other words, the implementation backs the desiderata of supporting *unanticipated* extension and modification, as seen for instance in Section 11.3.5.

The implementation utilizes Groovy AST transformations to transform occurrences of `new` statements that refer to template classes into reflective runtime calls. An extended Groovy parser is utilized to transform templates into an AST representation that can subsequently be manipulated by user implementations of the meta-

level interfaces. The tiny internal DSL for instantiations is implemented by utilizing Groovy closures for which the execution is delayed until they are assigned a specific delegate that overrides the `methodMissing` and `propertyMissing` meta-object protocol methods. Applying this in a builder-like fashion (resembling e.g. the Groovy `MarkupBuilder`), the framework dynamically builds a tree representing the requested instantiation at runtime, including any elements added to the DSL by strategies as exemplified in Section 11.3.5.

Performance While performance tuning has not been a main activity in our work with GPT thus far, and our testing in this area is far from comprehensive, it seems reasonable to include a few words on the topic. All tests were run on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor and 4 GB RAM. The operating system was Mac OS X 10.6.4, and all the tests were run from a standalone Groovy script file in the NetBeans IDE [23] version 6.7.1, with Groovy version 1.7.1 and JVM version 1.6.0_20.

Object creation from template classes, i.e. statements of the form `new INST_T.A()`, is in the order of 2 to 3 times slower than their plain Groovy counterparts. This is not surprising, since our approach is based on reflective `newInstance()` calls, while Groovy itself resolves plain `new` statements a compile-time.

Method call, method execution and field access to objects created this way incur no performance hits compared ordinary Groovy objects.

Template instantiations have no direct counterpart in Groovy with which a reasonable comparison can be made, since ordinary usage of the `import` statement is handled at compile time. However, we can at least give a few indications of the performance implications of runtime instantiations. On our test setup, the runtime instantiation of a single simple (containing just a few classes) template took in the order of 30 to 40 ms. This includes both the parsing and modification done by GPT, and the application of the Groovy compiler itself. However, this time will increase in a linear fashion with the number of classes, and for complex templates the instantiation time might quickly become a concern for real applications. However, it should be noted that no attempts at optimizing the code for instantiations have been performed at present.

When meta-level strategies are involved, an important factor will of course be what the actual code in these strategies does. However, utilizing rather simple strategies, e.g. such as the one in the Active Object example from Section 11.3.6, during instantiation does not introduce any overhead that we have been able to reliably measure.

11.6 Related Work

Virtual classes were introduced in the BETA language [21, 22]), and have subsequently been utilized in a number of mechanisms, such as e.g. gbeta [8], J& [24] and Caesar [1] (the latter also contains AOP features, see below). The main idea is that an outer class encloses one or more inner classes that are virtual in the sense that they can be overridden by defining new versions in subclasses of the outer class. This provides

some of the same flexibility for unanticipated adaption as GPT; a main difference, however, is that in GPT this is all handled during instantiation, and there need not be any sub/superclass relationship between adaptor and adaptee. J& and gbeta support multiple inheritance, which is something that GPT does not allow. However, for code composition, we can attain some of the benefits of multiple inheritance by merging template classes. In NewSpeak [4], the *only* modularity construct is the class, and nesting is applied to provide a module hierarchy. All class references are virtual, and thus every class can function as a mixin. Like GPT, NewSpeak allows for multiple independent instantiations of the same module. A module can be adapted and extended by creating subclasses of outer and inner classes, and overriding virtual definitions. However, since GPT is based on templates whose classes become ordinary classes only after an instantiation, the range of allowable modifications can be greater than in typical virtual class-based mechanisms.

Subject-Oriented Programming (SOP) [14] is an approach that allows multiple subjects to have differing, possibly incomplete, views of objects in a system. A subject represents a view of the world, i.e. a subjective perspective, as manifested by state and available behavior. The stated primary goal of the mechanism is to *"facilitate the development and evolution of suites of cooperating applications."* [14, page 412]. Multiple subjects can be composed to allow them to interact, and to react to each other's behavior. Subject composition is governed by composition rules that may be general or specific, though [14] explicitly leaves the formalization of such rules for later work.

In [25], a model for SOP composition rules is described, relying on *labels* that describe the declarations in a (flattened) subject. Composition clauses are then defined for such labels, which again may be used to define more high-level composition rules.

GPT classes, and thus package templates, are declarationally complete, and do not, as opposed to SOP subjects, *inherently* represent a subjective and incomplete view of the global system. Rather, templates are to be thought of as complete units in their own right, that *may* be composed with other units to form a larger whole. SOP does not define a model for runtime (or, for that matter, compile-time) interaction with composition rules.

Aspect-oriented programming (AOP) [17] can be seen as a special case of meta-level programming in the sense that meta-level entities (the aspects) make changes to base-level code, and the base-level is *oblivious* to such changes. However, the changes that an aspect is allowed to perform are typically quite limited compared to the meta-level of GPT. For instance is renaming of classes or methods typically not allowed, nor is class merging. On the other hand, aspects have the ability to impact an entire application (they are typically global in scope), while the applications of GPT's meta-level are typically quite narrow.

In [29], Tanter describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects), however, these aspects may, as opposed to runtime instantiations in GPT, affect behavior only, and not class structure or hierarchy.

With a Meta-Aspect Protocol [6], the semantics of an aspect-oriented language is in the hand of the programmer, allowing for control over e.g. advice ordering, aspect in-

teractions etc. While the subject matter is somewhat different (i.e. a meta-level protocol for AOP instead of module composition), the concrete approach in that paper, which also includes an open implementation in Groovy, was indeed an important inspiration for our work.

The **Groovy** language itself contains a powerful feature in its support for compile-time **AST transformations**, and we utilize this in our implementation to a certain extent in order to transform statements relying on classes generated at runtime by an `instantiate` method call. The implementation of the meta-level API presented herein also relies on transforming (template) ASTs, however, the built in AST transformations in Groovy are not flexible enough on their own to provide the level of customizability that we need in our framework. In particular, the different phases supported by our interception points would be very hard (if at all possible) to implement in that way. Furthermore, our framework provides a much higher degree of abstraction, and supplies meaningful meta-level constructs as opposed to just directly providing the means to manipulate the AST.

11.7 Concluding Remarks and Future Work

We have shown how module composition mechanisms, as exemplified by the GPT mechanism, can support varying composition semantics through the use of meta-level strategies that exploit an API that gives access to composition/instantiation specifications and provides hooks at specified points throughout the instantiation process. Strategies can be applied either globally, or within explicitly defined scopes. This further entails that a program can contain a variety of different composition semantics that can even be changed throughout the lifetime of a program, based on runtime conditions.

We have also presented an approach that allows meta-level code to be encapsulated within a module, enabling the module to take control of its own instantiation. The API through which meta-level functionality is accessed is easily extensible, allowing the user to e.g. add new keywords for new functionality in instantiation specifications.

Directions for future work include creating a more robust and performant implementation, and to carry out real-world experiments with the mechanism. Furthermore, we would like to explore meta-level strategies also for the statically typed version of PT as well as for the dynamic version. Work is in progress with respect to supporting statically typed runtime instantiations [28], and strategies (both runtime and compile-time) seem to hold interesting possibilities.

Acknowledgements

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30). The authors would like to thank Fredrik Sørensen for many interesting and insightful discussions pertaining to this and related topics, and Vebjørn W. Axelsen for valuable feedback on the

manuscript. Finally, we would like to thank the anonymous reviewers for many suggestions on how to improve the content and presentation of this paper.

Bibliography

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] E. Axelsen and S. Krogdahl. Groovy package templates - supporting reuse and runtime adaption of class hierarchies. In *Proc. Dynamic Languages Symposium '09*, 2009.

[3] E. Axelsen and S. Krogdahl. Towards pluggable design patterns utilizing package templates. In *Proc. Norsk Informatikkonferanse '09*, 2009.

[4] G. Bracha, P. V. D. Ahé, V. Bykov, Y. Kashai, and E. Mir. Modules as objects in newspeak. In *ECOOP 2010: Proc. 24th European Conference on Object Oriented Programming*, 2010.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[6] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2009. ACM.

[7] R. Eckstein. Mixins in JavaFX 1.2 technology. Sun Microsystems, URL: `http://java.sun.com/developer/technicalArticles/javafx/mixin/`, 2009.

[8] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[9] R. Field. JavaFX language reference. Sun Microsystems, URL: `https://openjfx.dev.java.net/langref/`.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[12] The Groovy homepage. URL: `http://groovy.codehaus.org`.

[13] Groovy: Runtime vs compile time, static vs dynamic. URL: `http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic`.

[14] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[15] G. Kiczales. Beyond the black box: Open implementation. *IEEE Softw.*, 13:8–11, January 1996.

[16] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[18] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[19] G. R. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[20] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[21] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[22] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[23] The NetBeans homepage. URL: `http://netbeans.org/`.

[24] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, 2006. ACM.

[25] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. *SIGPLAN Not.*, 30(10):235–250, 1995.

[26] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 734–737, New York, NY, USA, 2000. ACM.

[27] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[28] F. Sørensen, E. W. Axelsen, and S. Krogdahl. Dynamic composition with package templates. In P. Lahire, editor, *Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*. CEUR-WS.org, 2010.

[29] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, 2008. ACM.

Chapter 12

Paper V: Challenges in the Design of the Package Template Mechanism

Authors. Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl and Birger Møller-Pedersen

Publication. Special issue on Modularity Constructs in Programming Languages, Transactions of Aspect-Oriented Software Development, 2012.

Abstract. Package Templates is a mechanism for writing modules meant for reuse, where each module (template) consists of a collection of classes. A package template must be instantiated in a program at compile-time to form a set of ordinary classes, and during instantiation, the classes may be adapted by means of renaming, adding attributes (fields and methods), and supplying type parameters. Also, classes from two or more instantiations may be merged to form a new class which will have all the attributes from the merged classes and also the attributes added in the instantiating template. An approach like this naturally gives rise to two distinct dimensions along which classes can be extended. One is the ordinary subclass dimension, while the other is comprised of the ability to merge classes and to add attributes during instantiations. The latter dimension thus allows for a form of static multiple inheritance and adaption, that is handled entirely at compile-time. This paper discusses how these dimensions play together in the different mechanisms that make up the Package Templates approach, and the design considerations involved. The paper also argues that the compromise in Package Templates between simplicity of the type system on the one hand and expressiveness on the other is, for most purposes, better than similar approaches based on virtual classes.

Keywords. Programming Languages, Language Constructs, Object-Oriented Programming, Modules, Packages, Modularization, Inheritance, Templates

12.1 Introduction

The basic concepts of object orientation, that is classes, subclasses including subtype polymorphism and virtual methods, were introduced with the Simula language in 1967 [15]. These concepts became important for later research concerning how languages should be designed to support specialization/generalization, separation of concerns, reuse of code, etc. Since 1967, a number of mechanisms have been added to this original repertoire of language mechanisms, e.g. multiple inheritance [40], generic classes [8], virtual classes [30, 31], aspects [27], traits [37], and mixins [9]. Some have focused on specialization/generalization, others on separation of concerns and reuse of code, and some have tried to combine these.

Package Templates, or PT for short, is another such mechanism, with emphasis on separation of concerns and reuse; its main ideas have been presented in [28], while some of the issues discussed in this paper have previously been presented in [38]. PT is intended to be viable for inclusion into various programming languages, so the mechanism is therefore more a set of ideas and concepts that must be adjusted to each host language, rather than a fully defined language. It is here (and in [28]) presented as an extension to Java, with syntax and solutions tailored for that language. An implementation of PT for and in the dynamic language Groovy is explored in [2, 3].

When designing a mechanism such as PT, there are obviously many issues for which a choice between diverging design considerations needs to be taken, and where which alternative is the best is not obvious. The aim of this paper is thus both to discuss some of the design issues we had to deal with during the development of PT, and furthermore to provide a more thorough exposition of some of the issues that were described in less detail in [28, 38]. This entails that we will discuss issues that we believe we have solutions for, but also issues that we are still actively researching. In addition, we discuss some extensions to PT that are being considered.

The modules (templates) in PT are collections of classes. The main motivation for this is that concepts of some complexity are usually defined in terms of other concepts and relations between these concepts. A well-known example is the concept of a graph, which typically will be defined in terms of concepts like nodes and edges, and mechanisms for handling structures of these. Utilizing an object-oriented approach, such a complex concept can naturally be reified as a collection of classes that represent the constituent concepts (here *Node* and *Edge*), and a name for the collection as a whole (here *Graph*). The collection may e.g. be represented in a program by an enclosing class (if nested classes are supported), or by a Java-like package.

In order for a mechanism for handling such class-collections to be useful, it would be preferable if we could make suitable adaptions to them when they are used in a program. We may e.g. want to rename the classes of the collection and to add attributes (methods and fields) to the classes. We could e.g. use the Graph for making graphs of Cities and Roads. It would then be desirable to rename `Node` to `City`, and `Edge` to `Road`; in addition `Road` should e.g. have a `length` attribute and `City` a `name` attribute. One might think that this easily can be obtained by defining, in the program, new subclasses of the classes from the original class-collection. However, this would

not give the effect we want in PT. Assume that the collection `Graph` has the following two classes:

```
class Node { ... }
class Edge {Node from, to; ...}
```

In addition there will be code in each of these classes that may include creation of objects of the classes `Node` and `Edge`, and the effect we want here is that when this class collection is used to form classes `City` and `Road`, a statement like `new Node()` would automatically be changed to `new City()`. However, if `City` and `Road` are simply subclasses, we will not get this effect. Furthermore, given a variable `Road someRoad` in the program, we would like the following assignment to compile without a cast: `String someName = someRoad.from.name`. However, if we use ordinary subclasses, a statement of the following form would be required: `String someName = ((City)someRoad.from).name`. Finally, if the classes `Node` and `Edge` had the common superclass `GraphElem`, and we also wanted to extend this to e.g. `GeographyElem`, then `City` and `Road` would not be subclasses of this class. To summarize the problem, we would like an instantiation to preserve the hierarchy and other relationships between the classes, and at the same time be able to modify them.

One approach to class-collection mechanisms that remedies much of the above problems is based upon *virtual classes*, a mechanism pioneered by BETA [30]. This approach uses an enclosing class for representing the class-collection itself, and *inner* (nested) classes to represent the constituent concepts. The inner classes that should be subject to adaptions are defined as virtual classes. To get different adaptions of the basic set of inner classes one then defines subclasses of the enclosing class, and in these subclasses overrides (or rather *extends*) the inner virtual classes. Thus, within objects of the subclasses of the enclosing class, one will have different adaptions of the virtual inner classes. Typical representatives for such approaches are gbeta [18], Caesar [1], J& [34, 35], and Newspeak [10].

The virtual class mechanism entails that generation of objects of the virtual classes in the original outer class will imply generation of objects of the redefined classes, and explicit casts are thus not needed. Usually, it is possible to extend the virtual classes with new attributes and methods, but there are no means for renaming. A drawback with the virtual class mechanisms is that, at least in statically typed languages, they tend to require rather complicated type systems, with concepts such as prefixed types, exact types, dependent types, and intersection types [13, 21, 35]. However, such mechanisms provide a large degree of dynamic expressiveness, which can be valuable in certain situations. They also show a good "economy of concepts", in that they almost solely use the traditional concepts of object-orientation (including nested classes), only adding the concept of virtual classes and in some cases multiple inheritance. To some extent PT represents a move away from this economy principle in that it introduces a brand new mechanism for handling class-collections.

PT is based upon (Java-like) packages. However, the options for unintrusive adaption of Java packages to their use are limited. Therefore, we have added a "generic level" to the package mechanism, and the result is *Package Templates*. Thus, a package

template (or just *template* for short) syntactically looks much like a Java package. While Java packages may only be imported as they are, a package template must be *instantiated* (at compile-time) in a program before the classes of the template become ordinary classes in that program.

A template may be instantiated multiple times in the same program, and each instantiation will produce a new, independent, set of the classes declared in the template. Also, each instantiation may employ adaptions that are required in order for the new classes to fit well with their use in the program. Adaptions may include adding attributes (fields or methods) to the classes, renaming of declarations in the template, overriding of abstract or virtual methods defined in the template classes, and providing actual types to the type parameters of the template.

Thus, a main idea with PT is that the manipulations of class-collections that can be described in PT should be taken completely care of during compilation. To form the resulting ordinary classes of an instantiation, a copy is first taken of the template classes, and then all occurences of the class names defined in the template (e.g. `Node` above) will be replaced by the name of the extended class (here `City`), thus obtaining the ”virtual class effect” we want. This also has the effect that the classes described in the templates will not be present at run-time, and we will not have objects of these. This is the main reason why PT gets a simpler type system than what is typical for virtual class systems.

The templates of PT may seem similar to those of C++, as they both can be seen as advanced versions of traditional textual macros. However, templates in PT are fundamentally different from their C++ counterparts in the sense that PT templates can be fully type checked as independent units, without considering their subsequent usage in a program.

With PT, programmers have the option to use compile-time composition of template classes instead of subclassing for pure code reuse, and may thus to a large extent restrict their use of subclassing to classification purposes. PT also has a *merging* mechanism, by which we, during instantiations, can merge two or more classes from different template instantiations to a new class that gets the union of the attributes and a name given by the programmer. Then, all types referring to these classes in the involved template instantiations are unified, under this name. And as before, the original names of the merged template classes will not exist at run-time. This mechanism may be seen as a form of ”static multiple inheritance”, and is meant for compile-time composition of code from different sources. Our hope is that this mechanism can take care of many of the situations where general multiple inheritance is used today. Thus, for pure code reuse there should be no reason to introduce traditional multiple inheritance

The rest of this paper is organized as follows: Section 12.2 gives an overview of PT. Section 12.3 comprises the main content of this paper, and discusses a number issues involved in the design of PT, what problems arise and how they may be solved. Related work is discussed in Section 12.4, and Section 12.5 gives some concluding remarks.

12.2 An overview of PT

In this section we give a general overview of the PT mechanism. More details will be given where relevant in Section 12.3.

12.2.1 Basics

For definitions of package templates, we shall use the following syntax, where class definitions are enclosed by a `template` construct:

```
template T < ...compile-time type parameters... > {
   class A { B b; ... }  // b is referred to in the text below
   class B { ... }
}
```

We will return to the template type parameters in Section 12.3.7, and will hence not cover them in this section.

The classes inside the template `T` are referred to as *template classes*, and can essentially be written as ordinary Java classes. For the template classes `A` and `B` to become ordinary classes we have to instantiate the template `T` in a package `P`. This is done by the `inst` statement, as shown in the second line in the example below. This statement makes the template classes `A` and `B` available in `P`, but here under the names `AA` and `BB`, respectively, as specified by the arrows (`=>`) in the *with clause*.

Additions to the template classes are given in class-like constructs called *addition classes*. Such addition classes are provided for `AA` and `BB` as explained below, while `C` and `D` are new classes in the package `P`. Note that we can refer to `C` and `D` in the additions to `AA` and `BB`. Throughout this paper, we shall write packages in the same style as we write templates, i.e. with their contents enclosed by curly braces, as shown below:

```
package P {
   inst T with A => AA, B => BB;
   class AA adds { ... additional attributes in AA ... }
   class BB adds { ... additional attributes in BB ... }
   class C { ... }
   class D { ... }
}
```

When a template is instantiated, its contents can be adapted. The most important forms of such adaptions are the following:

- Elements of the package template may be renamed. This is done in the with-clause of the `inst`-statement, and is here only shown for class names. For renaming of class attributes another arrow is used (`->`), so that if we also want to change the name `b` to `bb`, the line must read: `inst T with A => AA (b -> bb), B => BB;`. Note that all renaming in PT is done in a "semantic way". That is, renaming is done based on the name bindings from the static analysis.

- In each instantiation the classes in the template may be given additions: fields and methods may be added and virtual methods may be redefined. This is done in `adds`-clauses as shown above. The order of the `adds`-clauses and other declarations in the package is not significant.
- We may "merge" classes from different template instantiations.
- If the template has formal type parameters, actual type parameters must be supplied.

Note that the additions are made in a "static virtual" manner. That is, if we in the `adds`-clause of `BB` declare `int i`, we may in the `adds`-clause of `AA`, without casting, write `b.i = 1;` even if the variable `i` was not known when `b` was declared in `A`.

As an example we again consider a template for defining graphs:

```
template Graph {
   class Node {
      Edge[] outEdges;
      Edge insertEdgeTo(Node to){ ... }
      ...
   }

   class Edge {
      Node from, to;
      void deleteMe(){ ... }
      ...
} }
```

Below, the template `Graph` is instantiated in the package `RoadAndCityGraph`, in order to use objects of class `Node` for representing cities and objects of class `Edge` for representing roads (and we disregard, for simplicity, the fact that template `Graph` defines directed graphs). For this purpose, `Node` is renamed to `City`, `Edge` to `Road`, and both get additional attributes. We also rename `insertEdgeTo` to `insertRoadTo`. When renaming a method, its parameter types must be given, since there (in general) may be definitions of method overloads or a variable with the same name.

```
package RoadAndCityGraph {
   inst Graph with
      Node => City (insertEdgeTo(Node) -> insertRoadTo),
      Edge => Road;

   class City adds {
      String name;
      void someMethod(){
         int n = outEdges[3].length; // 1, see text below
         City c = ... ;
```

```
         Road r = insertRoadTo(c);    // 2, see text below
   } }

   class Road adds{
      int length;
      void someOtherMethod(){
         String s = to.name;          // 3, see text below
 ...
} } }
```

Here, a copy is first made of the Graph classes, then the specified renamings are done in these, and finally each class gets the additions given in the corresponding `adds`-parts. The `inst` statement and these `adds`-parts are then removed, and replaced by the result of the above transformations. Thus the names `Node`, `Edge`, and `insertEdgeTo` will not exist at all in the package RoadAndCityGraph, and the statements marked 1, 2, and 3 above will all be legal, and this can be determined statically. No objects of classes `Node` and `Edge` will ever be generated in the package `RoadAndCityGraph`, as occurrences of e.g. `new Node()` in the template are changed to `new City()`. The method `insertEdgeTo` in `Node` will have the following signature within `RoadAndCityGraph`:

```
Road insertRoadTo(City to){ ... }
```

Virtual or abstract methods defined in a class within a template can be overridden by methods defined in an `adds`-clause as part of an instantiation. That is, if the addition to class `City` has a definition of the method `insertRoadTo` with the same signature as above, then this will override the method `insertEdgeTo` in `Node`.

12.2.2 Subclass Hierarchies Within Templates

PT allows ordinary subclass hierarchies to be defined inside templates. As part of an instantiation of a template, all classes in such hierarchies (and not only the leaf classes) may be given additions. As an example, consider the following sketch of a template:

```
template Vehicles {
   class Vehicle { float maxSpeed; ... ; }
   class Car extends Vehicle { int numOfSeats; ... ; }
   class Truck extends Vehicle { int length; ... ; }
}
```

and a use of it:

```
package TrafficSimulation {
   inst Vehicles with Vehicle=>SimVehicle,
      Car=>SimCar, Truck=>SimTruck;

   class SimVehicle adds{ PosType position; float curSpeed; ... ; }
   class SimCar adds{ int luggageVolume; ... ; }
   class SimTruck adds{ int loadCapacity; ... ; }
   ...
}
```

Note here that `SimCar` and `SimTruck` automatically will be subclasses of `SimVehicle`, as this is true for the corresponding classes in `Vehicles`. It is perfectly legal here to e.g. add an abstract method in `SimVehicle`, and give concrete versions of it in `SimCar` and `SimTruck`. But what if an abstract method is defined in `Vehicle` and is given concrete versions in both `Car` and `SimVehicle`? Which version will then be used if it is called from `SimCar`? These and similar questions will be discussed in Sections 12.3.2 and 12.3.4.

12.2.3 Multiple Instantiations

A basic idea in PT is that one can do two or more instantiations of the same template in the same scope, so that one e.g. can use the `Graph` template for producing the classes `City` and `Road` as above, and in the same scope use it to represent e.g. the structure of pipes and connections in a water distribution system.

12.2.4 Instantiation of Templates Within Templates

Templates are always written as independent entities, and are instantiated at the outermost level of packages or other templates. This makes it possible to build hierarchies of instantiations, which, for some tasks, turns out to be very useful. As an example, assume that we also want the package `RoadAndCityGraph` from the beginning of Section 12.2 to be a template, so that further additions can be made to the classes `City` and `Road` before we use them at a later stage. The only change needed is to use the keyword `template` instead of `package` when defining `RoadAndCityGraph`. Then this template can be instantiated in a package e.g. for drawing maps where also positions are needed for cities.

When templates are instantiated within templates, as for the `RoadAndCityGraph` template outlined above, this should work as follows: Whenever `RoadAndCityGraph` is instantiated, instantiations specified inside of this template should also be carried out. Cyclic structures of templates instantiating templates are not allowed.

12.2.5 Merging Template Classes as Part of Instantiations

In order to get optimal reuse of code, it is important to be able to merge independently written code. A challenge here is to merge the independent types of the code compo-

nents to one such that they can cooperate in a type safe manner. PT's answer to this is to offer a mechanism where a class from one instantiation is merged with a class from another to form one new class. Code from the different templates can only access the attributes of objects of the new class that stem from the respective templates, while in the scope with the instantiations and the merge, all attributes of the new class can be accessed.

Syntactically, merging is obtained by allowing classes from two or more template instantiations to share a common addition class, and they thereby end up as one class, with the name of the addition class. The new class gets all the attributes of the instantiated classes, together with the attributes of the addition class.

The following example illustrates how this mechanism works: As in the example in Section 12.2.3, we assume that we have the template `Graph`, and that we want to use this as a basis for forming classes `City` and `Road`. However, instead of adding the extra attributes of `City` and `Road` in the `adds`-clauses, we this time assume that we also have another template `GeographyData` with classes `CityData` and `RoadData` where the extra attributes to form the classes `City` and `Road` are defined:

```
template GeographyData {
   class CityData{ String name; ... ; }
   class RoadData{ int length; ... ; }
}
```

We can now define the classes `City` and `Road` as merges of `Node` and `CityData`, and `Edge` and `RoadData`, respectively, with addition classes `City` and `Road`. We do this by instantiating the templates as follows:

```
package RoadAndCityGraph2 {
   inst Graph with Node => City, Edge => Road;
   inst GeographyData with CityData => City, RoadData => Road;
   class City adds{ ... }
   class Road adds{ ... }
   ...
}
```

As described above, classes that are renamed to the same name, such as `Node` and `CityData`, will be merged to one class, under the new name. The resulting class `City` will then have all of the attributes of `Node` and `CityData`, plus those given in the shared addition class, and likewise for the new class `Road`. Objects of these classes can be accessed in exactly the same way as with our earlier definitions of `City` and `Road`.

Related to this construct, there are obviously many issues that we need to resolve. We can for instance easily get name collisions, and what about abstract or virtual methods defined in two or more of the merged classes? Similarly, what about constructors? These and other issues are discussed in more detail in Sections 12.3.2, 12.3.3 and 12.3.5.

12.2.6 Multiple Inheritance

Through merging, PT gets a form of static multiple inheritance that is handled entirely at compile-time (that is, the classes that are merged to form the new class are not accessible nor present at runtime in the final program). However, we do not want this static form of multiple inheritance to also imply traditional multiple inheritance. We can easily forbid explicit multiple inheritance in templates and programs. However, even if we do this, we can indirectly get multiple superclasses to a class through merging, and we therefore have to introduce further restrictions. These issues are addressed in Section 12.3.6.

12.2.7 Interfaces and Inner Classes

Interfaces are mentioned a few places in this paper where they are relevant, but in general the full role of interfaces in PT is an important topic for future work, and not discussed in detail in this paper. The same is true for inner (nested) classes.

12.3 Discussion of Main Challenges

12.3.1 Template Classes can be Extended Along Two Dimensions

Standard object-orientation provides a single "dimension of extension" for classes, which is traditional subclassing. By introducing templates and `adds`-clauses as parts of instantiations, template classes get a second dimension of extension. This dimension has other properties, that represent both new possibilities and new challenges.

An important design consideration is whether both of these dimensions can be referred to in the code, i.e. if the addition dimension is reified through the available language primitives, or if it is only present on a more conceptual level. For instance, providing the ability to refer to overridden method definitions in a template class from a package class would require specific syntax and semantics.

Thus, an immediate observation is that the *flattening property* as employed by traits [37] cannot coexist with a reified two-dimensional structure; the flattening property entails that the semantics of the composed unit is exactly the same as if the contents of every constituent was written directly in the composed unit itself.

Designing a version of PT supporting the flattening property could in itself be an interesting endeavor, and a simpler, yet still useful, subset of what we will discuss in the rest of this paper would be worth exploring. However, supporting this property would also seemingly preclude constructors in templates (see Section 12.3.5) as well as calls to overridden template class methods etc.

For PT we have found, through the examples we have studied, that e.g. being able to explicitly refer to overridden methods defined in a template class from a package class does indeed seem very useful, and we have thus opted for introducing explicit ways of referring to each dimension of extension.

General Overview.

To see the general scheme, we look at the following sketch of a program with two templates and a package; for simplicity we keep merging out of the picture for now.

```
template T1 {
   class A1 { }
   class B1 extends A1 { }
   class C1 extends B1 { }
}

template T2 {
   inst T1 with A1 => A2, B1 => B2, C1 => C2;
   class A2 adds{ }
   class B2 adds{ } // Subclass of A2, as B1 is subclass of A1
   class C2 adds{ } // Subclass of B2, as C1 is subclass of B1
}

package P {
   inst T2 with A2 => A3, B2 => B3, C2 => C3;
   class A3 adds{ }
   class B3 adds{ } // Subclass of A3
   class C3 adds{ } // Subclass of B3
}
```

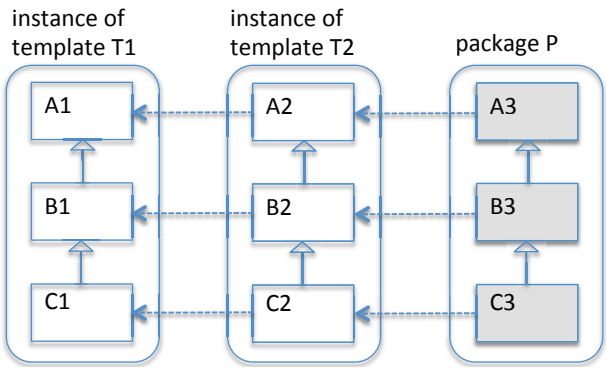

Figure 12.1: Template classes can be extended along two dimensions, the *subclass dimension* and the *addition dimension*. Note that the figure depicts instances of the templates `T1` and `T2`, respectively. `P` is an ordinary Java package.

Figure 12.1 shows a graphical representation of the relationships between the different parts of the program sketch above. Extensions in the traditional dimension of

sub/superclasses are drawn along a vertical axis (with arrows from subclass to superclass), while extensions in the new dimension of adding attributes during instantiations are drawn along a horizontal axis (with a dotted arrow from the `adds`-clause to the template class). We shall call these dimensions the *subclass dimension* and the *addition dimension* respectively, and a class at the head of the arrow is called the superclass and the *tsuperclass*, respectively, of the class in the other end. Thus, in Figure 12.1, we can for instance see that `C1` is a tsuperclass of `C2`, and `B2` is a superclass of `C2`. The contents in `A2`, `B2`, `C2`, `A3`, `B3` and `C3` will (as seen in the code above) occur syntactically as `adds`-clauses, while `A1`, `B1` and `C1` will look like normal classes or subclasses.

(A note on syntax: we have chosen not to write e.g. `class B2 extends A2 adds { ... }`, and thus the subclass relationship between `A2` and `B2` in `T2` is implicitly given only by the relationship between `A1` and `B1` in `T1`. We do acknowledge, however, that there are valid arguments for both options here.)

There are certain essential differences between the two dimensions:

- The most obvious difference is the following: The subclass dimension is "dynamic" in the sense that the `A`, `B`, and `C` levels will exist at runtime so that we can generate objects of all three of the classes `A3`, `B3` and `C3` (in gray above), and that these classes are traditional subclasses of each other. The addition dimension, on the other hand, is more static in the sense that it is taken care of entirely at compile-time. Classes `A1`, `A2`, `B1`, `B2`, `C1`, and `C2` will not exist as separate entities at runtime, but rather as parts of `A3`, `B3` and `C3`, respectively; thus, even if we may specify `new B2(...)` in a method of a class in template `T2`, it will be transformed to `new B3(...)` in the package `P` by the compiler.

- Another important difference is that the subclass dimension only has single inheritance, while the addition dimension enjoys a sort of multiple inheritance through merging. Thus, everything that has to do with this special form of multiple inheritance is treated at compile-time, and at runtime we will only have straight-forward single inheritance. We think this is a promising compromise between the need to combine code from different sources and adjust it for reuse purposes, and the complexities of traditional multiple inheritance.

Thus, these dimensions give us a number of possibilities, but they indeed also give rise to problems for the design of a consistent system. Some of these problems have rather straightforward solutions, while others (e.g. those concerning constructors) are more challenging. These problems will be discussed in the subsections below.

12.3.2 Virtual Methods and `super` Calls

With two dimensions, method call resolution is not as straightforward as for a single dimension. Assume that `A1` from Figure 12.1 defines an abstract method `m()`, and that an implementation is provided both in the subclass `B1` and in the addition class `A2` (and nowhere else). Given a call to `m()` e.g. in `B2`, which version will be chosen in an object of the class `B3`? (Note that there cannot be any objects of any classes except `A3`,

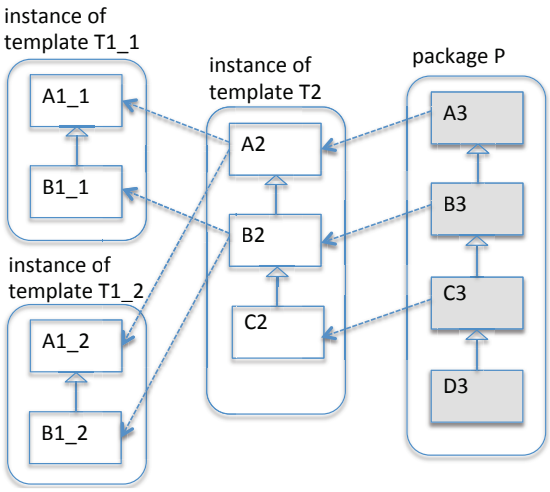

Figure 12.2: An example of a more complex instantiation.

`B3` and `C3` in this program.) As a guide in this and similar cases, we emphasize the fact that the addition dimension represents compile-time composition to a single class, while the different levels of the subclass dimension still exist at runtime. This entails that the addition dimension should bind strongest, so that the version in `B1` should be chosen.

Now, what if the method is also overridden in `C2` and `B3`, and the call is done in a `C3`-object (but still from the program text of `B2`)? As for calls to (virtual) methods in most languages, the version chosen should only depend on the class of the object, and according to the binding priority indicated above, we should look for the correct version by first searching along the addition dimension from the class of the object, and if it is not found there, move one step up in the subclass dimension, and repeat the process. Thus, even if the call is made from the text of `B2`, the version in `C2` will be chosen.

Merging and Complex Instantiations.

The preceding example possesses a somewhat artificial regularity in that it contains an equal number of classes (3) in each template/package. In the general case this is obviously not necessarily so. Figure 12.2 shows an example of a more complex instantiation that includes merging and templates that have different numbers of classes (a sketch of the program is omitted, but code similar to the previous example is hopefully easy to envision).

When merging is involved, a given class will have two or more tsuperclasses, and this is something that we need to take into consideration when resolving method calls.

Note that if the classes that are merged have superclasses, such as e.g. `B1_1` and `B1_2` in the figure, these superclasses (i.e. `A1_1` and `A1_2`) will also have to be merged; more on this in Section 12.3.6.

A pseudo-code algorithm for method body lookup for a virtual method call is shown below. At the start of the algorithm, `C` is the class of the object in which the call occurs, and `m` is the signature at the call site. As we will get back to in Section 12.3.3, all name conflicts must be resolved for the classes that are to be merged, and thus the algorithm will never have to choose between several compatible method implementations at the same level, even when there are multiple tsuperclasses. Furthermore, all renaming must be performed in the templates before running the algorithm; when the algorithm searches for a method, it will always use the names as they appear in the package.

```
MethodDefinition virtualLookup(MethodCallSignature m, Class C) {
 if(there is a local definition mdef matching m in C)
  return mdef;

 // A class will have more than one tsuperclass if a merge is involved, thus
 // foreach is used below. The order in which the tsuperclasses are processed is
 // not significant, and thus left as an implementation detail.
 foreach(direct tsuperclass AC of C along the addition dimension) {
  mdef = virtualLookup(m, AC)
  if(mdef != null)
   return mdef;
 }

 if(C is defined directly in a package &&
    C has a direct superclass EC along the subclass dimension) {
  return virtualLookup(m, EC);
 }
 return null;
}
```

Calls to super and tsuper.

Since PT has two dimensions of extension, the existing `super` keyword in Java is not sufficient to be able to reach all method definitions in a controlled manner. Thus, a mechanism for super calls along the addition dimension is needed. We will employ Java's traditional `super` keyword for reaching superclass methods in the normal subclass dimension, and introduce the new keyword `tsuper` for the new dimension.

When a template has regular inheritance inside it, methods can be overridden both in the `adds`-clauses and in subclasses. Below, we see an example of this (which is a slightly modified version of the example from Section 12.2.2).

```
template Vehicles {
  class Vehicle { String model; void display(){ out(model); } }
  class Car extends Vehicle {
    int passengers;
    void display(){ super.display(); out(passengers); }
  }
  ...
}

package RentalVehicles {
  inst Vehicles with Vehicle => RentalVehicle, Car => RentalCar;

  class RentalVehicle adds {
    int price;
    void display(){ tsuper.display(); out(price); }
  }

  class RentalCar adds {
    boolean nosmoking;
    void display(){ tsuper.display(); out(nosmoking); }
  }
  ...

    // In some method in some class:
    RentalCar rc = new RentalCar();
    rc.model = "Ford"; rc.price=120;
    rc.passengers=4; rc.nosmoking=true;
    rc.display(); // Prints "Ford 120 4 TRUE"
  ...
}
```

Inside the template `Vehicles`, the class `Vehicle` contains a virtual method `display`. This method is overridden in a normal Java manner in `Car`, in which the original implementation is called through the use of `super`. If the `Vehicles` template had been instantiated with no additions, it would work just as any ordinary Java package with regard to virtual methods and `super` calls.

However, here the method `display()` is also overridden in the addition part `RentalVehicle`, and in this method we want to call the original version in `Vehicle`. This can obviously not be done by means of `super` as `Vehicle` is not a superclass of `RentalVehicle`. Thus, the only way to reach a virtual method defined in a template class, but that has been overridden in an `adds`-clause to that class, is to use a `tsuper` call.

A pseudo-code algorithm for the lookup of a `tsuper.m(...)` call is shown below. When the algorithm is called, `C` is the class in which the call *syntactically* occurs (and

thus not necessarily the runtime class of the object, as opposed to for the previous algorithm), and `m` is the signature at the call site.

```
MethodDefinition tsuperLookup(MethodCallSignature m, Class C) {
 // A class will have more than one tsuperclass if a merge is involved,
 // thus foreach is used. The order in which the tsuperclasses are processed is
 // not significant, and thus left as an implementation detail.
 foreach(direct tsuperclass AC of C along the addition dimension) {
  if(AC has an accessible method definition mdef matching m)
   return mdef;
  else {
   mdef = tsuperLookup(m, AC);
   if(mdef != null)
    return mdef;
  }

 return null;
}
```

In the algorithm above we see how a call to `tsuper` will not result in any lookups along the subclass dimension. A missing method definition in the addition dimension will result in a compile-time error.

The lookup for a `super.m(...)` call is similar to the virtual method lookup presented earlier in this section. However, for `super` calls we do not start the lookup at the actual class of the object, but instead one step up from `C` in the `super`-direction. Thus, if a call to `super.m(...)` for instance occurs in `C2` in Figure 12.2, the search for `m` will start in `B3`.

An important thing to note is that these algorithms follow the general principle that an instantiation represents an actual substitution with semantic bindings. Each `tsuper` call will be transformed to an ordinary method call at compile-time. Of `tsuper` and `super`, only ordinary `super` calls are left in the final program, with the exact same semantics as a normal Java `super` call.

12.3.3 Name Conflicts

When classes are merged, there is a possibility that (accessible) attributes with the same signature are defined in more than one of the source classes, which gives rise to a conflict in the target class. One way to resolve such issues is to rely on ordering to resolve conflicts, such as e.g. giving the class from the template listed first in the instantiation precedence. Relying on order for conflict resolution is the approach taken with Mixin-based inheritance [9]. While this might have benefits in terms of ease of use etc, it might also lead to unwanted/unexpected behavior and problems with breaking changes in new versions of a given piece of code. Thus, for PT, we have opted for an approach that requires explicit conflict resolution. This is in many ways similar to the

way conflicts are resolved for traits [37]. Note, however, that as opposed to what is the case for traits, method exclusion is not supported by PT. Supporting this (in a safe manner) would require analyzing call graphs and potentially excluding/deleting all methods that refer to the method originally intended for exclusion. We believe that the extra complications involved with this outweigh the benefits.

This leaves two primary options for method conflict resolution: (1) to rename the conflicting method, or (2) to provide a new method that overrides all of the conflicting methods for a given signature.

Renaming (1) is the only resolution strategy that is currently supported in the implemented version of PT, and is done as a part of the instantiation as an optional clause to the `inst` statement, as explained in Section 12.2.1. Note that as opposed to method name aliasing in traits and similar systems, the method renaming in PT is "deep" in the sense that all references to the method in question (and only these) will be renamed for the current instantiation.

For the latter option (2), an override must be provided in the addition class of the merged classes. Typically, in such an override, one would want to call one or both of the existing methods. This can be achieved through `tsuper` calls, however, a mechanism for qualifying the original class names would be needed, e.g. as in `tsuper[A].m()`, where `A` is the name of the original class containing the desired implementation of `m`. However, for more complex scenarios, e.g. when the same template is instantiated multiple times, or different templates have classes with the same name, this simple qualification scheme is not sufficient to differentiate between the different classes. A more elaborate scheme could then be allowed, utilizing either template name or instantiation name (the latter is not treated explicitly in this paper, but would be relevant if the same template was instantiated twice) in addition to the class name.

The main reason for not supporting option 2 is the added complexity it brings, as discussed above, paired with the fact that there is not much to be gained by having this option available. For instance, if we really want one new method to be called whenever either of two existing methods was called prior to a merge, we could opt for the following solution:

```
// T has a class A with a method named m
inst T with A => AB (m() -> m_a);
// U has a class B with a method named m
inst U with B => AB (m() -> m_b);

class AB adds {
   void m_a() { // Overrides original definition from A
      m();
   }

   void m_b() { // Overrides original definition from B
      m();
   }
```

```
    void m() {
      // perform desired computations for calls to both
      // A.m() and B.m() here, for instance call
      // tsuper.m_a() and/or tsuper.m_b().
    }
}
```

For fields, providing an override is not an option, so the only solution for field conflicts is to rename one or more fields until there are no more conflicts.

Template classes may also extend ordinary Java classes and implement ordinary Java interfaces. In such cases, the rules for renaming are more restrictive, see Section 12.3.6 for details.

12.3.4 Abstract Methods in Template Classes

For a normal Java class, methods can be marked as `abstract` to specify that they need to be given a concrete implementation by subclasses. In PT, abstract classes can be defined both in packages and in package templates.

Along the addition dimension in PT, a similar concept can be defined for methods that are meant to be implemented in an addition class at a later stage. This is useful in order to allow template developers to "promise" that there will be an implementation of a given method, even if no sensible implementation can be provided when writing the template (e.g. because it will depend on its usage scenario). We shall mark such methods with the modifier `tabstract` (for *template abstract*); `tabstract` methods must be given an implementation in an addition class, at the latest when the enclosing template is instantiated in a package. Failure to provide such an implementation will result in a compile-time error.

An important difference between `abstract` and `tabstract` is thus that while one is not allowed to make objects (with `new`) of a class with `abstract` methods, one may do so with classes with `tabstract` methods. This is allowed since every such method will get an implementation at the latest when the template is instantiated in a package.

Below is an example using a combination of `abstract` and `tabstract` methods. `Figure` is an `abstract` class with an `abstract` method. No objects can be made from that class. The two subclasses are not abstract, and objects can be made from them since it is guaranteed that in a package using this template they will have implementations of the `draw` methods and thus be concrete classes. Thus, the requirement imposed by the `abstract` modifier can in fact be fulfilled by a `tabstract` declaration.

```
template Figures {
  abstract class Figure { abstract void draw(); }
  class Circle extends Figure { tabstract void draw(); }
  class Square extends Figure { tabstract void draw(); }
  ...
    Circle c = new Circle();
    Figure f = c;
    f.draw();
  ...
}
```

If a template containing classes with `tabstract` methods is used in another template, one may choose to implement the `tabstract` methods there or the implementation may be postponed until that template is instantiated in a package. Overriding a method that implements a `tabstract` method is allowed in the same way normal method overrides are allowed.

Since the mechanisms related to the `adds`-clauses of the templates are orthogonal to the mechanisms related to subclasses, they may also apply to `static` methods. This means that in PT one can declare a `static` method to be `tabstract`, and subsequently override a static method in an `adds`-clause and also call a static method defined in a template class with `tsuper`.

12.3.5 Constructors

Constructors are a convenient (and, in some languages, necessary) construct for initializing objects at runtime. In Java, all instance and static variables are initialized to neutral default values independently of the constructors. Thus, constructors are mainly a convenience for the programmer rather than a necessary construct to enforce type safety. Consequently, when designing a scheme for constructors in PT for Java, our main focus has been usefulness and comprehensibility for the programmer.

Classes in Java must have at least one (explicit or implicit) constructor, and (except for the effect of `this(...)` calls) exactly one constructor is invoked at each subclass level when an object is created. A constructor may use its parameters to initialize its own subclass level and/or pass them to a constructor of the superclass. The call to the constructor of the superclass must always come at the very start of any constructor so that all superclasses are initialized before the class itself. These rules create a regularity that as far as possible should be preserved in PT. An apparent idea is then to also allow template classes/`adds`-parts to have constructors, and in addition to using `super(...)` for calling constructors in the superclass introduce `tsuper(...)` for calling a constructor in the tsuperclass.

However, an obvious issue is that if constructors are called along both dimensions, a given constructor might generally be called a number of times in each class/`adds`-part. This is clearly not a desirable situation, and would would probably be very difficult to use in a controlled way if implemented.

One drastic alternative going in the opposite direction is to disallow constructors in template classes/`adds`-parts (and thus also the use of `tsuper(...)` calls) altogether, and say that initialization is the responsibility of the proper package classes/`adds`-parts. Only at that point can the developer have a complete picture of the workings of the fully composed classes. Note here that one can define a suitable set of ordinary methods (e.g. in template classes) that are designed to be called by proper constructors for initialization purposes, alleviating at least some of the problems with such an approach.

It would, however, be preferable if we could find a scheme where we could allow a constructor in each class/`adds`-part also in the templates, and where these are called in a systematic way so that exactly one constructor is called for each such part in a new object (again, except for the effect of `this(...)`). If we envision the general structure of programs like the one shown in Figure 12.1, such a strategy is not difficult to find. We might call it the "backwards E strategy", and it entails making `super` calls only in the rightmost column of the figure, and from there (after the `super(...)` call) make appropriate `tsuper(...)` calls towards the left (thus following the form of a backwards or mirrored E). Classes/`adds`-parts in templates should not have `super(...)` calls. Note that the compiler can syntactically recognize the rightmost column of such a program, as this will always be a package and not a template. Thus, the compiler can check that the constructors are called according to the rules. Parameters for constructors can be used freely, and we can easily make a scheme where default constructors and constructor calls are inserted if missing.

If we have merging, we must, in the `adds`-clause of the merged class, do one `tsuper(...)` call to a constructor in each of the merged classes, and here any order can be allowed. In these situations each `tsuper(...)` call should be qualified by a `[...]` construct as discussed for name conflicts in Section 12.3.3, e.g. like this: `tsuper[T.C](...)`, where `C` is a class that is being merged and `T` the instantiated template. Note, however, that as opposed to what is the case with name conflicts, the need for additional qualification in `[...]` cannot here be avoided by renaming or any similar scheme (unless we allow renaming of constructors).

To exemplify the "backwards E strategy", we look back to Figure 12.1, and now assume that a variable `a1` is defined in `A1`, `b1` is defined `B1` etc. To initialize these variables with constructors according to the scheme above, we could, in the different classes/`adds`-parts, have constructors as sketched below:

```
A1(a1){ this.a1 = a1; }
A2(a1, a2){ tsuper(a1); this.a2 = a2; }
A3(a1, a2, a3){ tsuper(a1, a2); this.a3 = a3; }
... B1 and B2 like A1 and A2 above ...
B3(a1, a2, a3, b1, b2, b3)
  { super(a1, a2, a3); tsuper(b1, b2); this.b3 = b3; }
... C1 and C2 like A1 and A2 above ...
C3(a1, a2, a3, b1, b2, b3, c1, c2 ,c3)
  { super(a1, a2, a3, b1, b2, b3); tsuper(c1, c2); this.c3 = c3; }
```

We can see that this will work out nicely for objects of the classes `A3`, `B3`, and `C3`. It is not difficult to set up a similar scheme for the more complicated case in Figure 12.2, but we leave that to the reader.

For this scheme to be complete, we must also consider object generation inside templates. In a template `T` with a template class `C`, we are allowed to write `new C(...)`. However, allowing this mandates that the final package class `PC` (given an instantiation of the form `inst T with C => PC`) supplies suitable constructors `PC(...)` for every parameterization of `C(...)`. Thus, a requirement to implement the necessary constructors is put upon the user of a template. However, we have not yet found the right balance between freedom of expression in templates and convenience for the user when instantiating templates, and our current implementation therefore only allows parameterless template class constructors to be called from within the template.

Except for the latter problem, the backwards E strategy might seem fine. It does, however, have one important drawback: When writing a class/`adds`-part in a template (e.g. in `B2` of Figure 12.1) one cannot directly control how the constructor of the superclass (here `A2`) is called. One is not even allowed to use `super(...)`, so the call to a constructor in the superclass will come indirectly "in from the right", through templates or packages that probably are not yet written. Thus, this prevents the templates from taking control over their own initialization.

An alternative approach could be labeled the "lying E strategy" (with "arms" and "legs" up), entailing that the constituents from leftmost template of Figure 12.1) are initialized before continuing to the right. This has the advantage that each template can allow its classes to use `super(...)` calls to initialize their superclasses, and thus making each template conceptually more self-contained. However, it turns out that also this strategy has its problems. First of all, it will necessarily involve breaking the principle that a superclass should be fully initialized before the class itself is. The second problem is that we now cannot determine syntactically where the "lowest row" is (corresponding to the rightmost column for the backwards E strategy), as this will depend on what sort of object is generated at runtime. If we e.g. look at the example from Figure 12.2, the lowest row (from right to left) corresponding to the different sorts of objects are as shown below. When merging is involved the merged classes (including their potential tsuperclasses) are listed one after the other, in no specific order.

```
In A3 objects:  A3, A2, A1_1, A1_2
In B3 objects:  B3, B2, B1_1, B1_2
In C3 objects:  C3, C2, B1_1, B1_2
In D3 objects:  D3, C2, B1_1, B1_2
```

It is not difficult to come up with a scheme where the constructors of all the classes in lowest row for the class of the created object are called. However, passing parameters to constructors in classes from this row is not entirely trivial, since there might not be a direct relationship between the runtime class of the object and the template classes that in the lowest row (as is the case for instance between `D3` and `C2` in Figure 12.2).

Thus, none of the proposals above seem to fully satisfy our needs, and we are continuing our research in pursuit of better solutions.

12.3.6 Avoiding Multiple Inheritance

PT supports merging of independently written template classes, and the class resulting from such a merge can thus indirectly get multiple superclasses if two or more of the merged classes have superclasses (other than `Object`).

Since we want PT to be a viable mechanism for languages that employ single inheritance, we need to introduce some additional rules, and consider a rather restrictive rule set to begin with. The first rule is that we forbid template classes to have superclasses (other than `Object`) that are defined outside the template itself and outside instantiations made in this template. Thus classes defined in an ordinary package cannot be used as superclasses. Note, however, that there are no such restrictions on implementing interfaces that are defined in this way.

However, we do not want such drastic restrictions for superclasses defined inside the same template (or inside instantiations made in this template), since this would disallow class hierarchies inside templates altogether, and we consider these very valuable. Thus, we also introduce the following rule:

If, in a set of instantiations, two or more template classes are merged, then the superclasses they might have (which, if they exist, are also template classes according to the first rule) must also be merged in the same instantiations. In addition it is required that such a merge must not result in a cyclic superclass-structure.

External Classes.

There are cases where the first rule (that template classes cannot have external superclasses) is obviously a nuisance, e.g. when we want to introduce our own exception classes in a template, that (in Java) have to be subclasses of those defined by the system. For these cases we introduce a special syntax to make the external relationship explicit and to ensure that multiple inheritance will not occur. When such an external superclass is used, the programmer must specify this explicitly with the keyword `external`, as in the following example:

```
package R{ class RC {...} }

template T {
   class A extends external R.RC {...}
}
```

This also has the consequence that the `adds`-clause to `A` in an instantiation of `T` (and transitively in `adds`-clauses of further instantiations) must also be marked as having an external superclass as follows:

```
template U {
   inst T with A => B;
   class B extends external R.RC adds {...}
}
```

```
template V {
   inst U with B => C;
   class C extends external R.RC adds {...}
}
```

Syntactically we can here omit `R.RC` (thus only indicating that `B` or `C` has *some* external, anonymous, superclass), but this will imply stronger restrictions in the following step. Now assume that we in a package `P` instantiate `V` and another template `W` containing a class `D` as follows:

```
package P {
  inst V with C => CD;
  inst W with D => CD;
  class CD adds { ... }
}
```

This merge will now be legal if class `D` in `W` has no external clause at all, or if the external clauses of `C` and `D` refer to the same external class. If one of the merged classes has an anonymous external clause the rule is stricter. Then no other of the merged classes can have any external clause at all. The `adds`-clause of `CD` does not need any external clause as a package class will not participate in any further merges.

Name Changes.

For instantiations involving classes that are subclasses of external classes, names stemming from an external superclass cannot be changed. This will never be a problem with respect to the use of renaming to obtain unique names in merged classes (since multiple inheritance is prohibited). However, when implementing external interfaces, name conflicts might be an issue. Thus, we propose that for instantiations involving classes that implement external interfaces, the names may be changed in the instantiation, but the resulting class will then no longer fulfill the requirements placed on the class by the interface. Unless the addition class subsequently defines the missing (due to the renaming) interface methods, there will be a compile-time error.

12.3.7 Templates with Parameters

Templates may have parameters, and in the paper describing basic PT [28] one kind of such parameters is discussed, which are type parameters working in much the same way as generic parameters to classes in e.g. Java or C#. However, recently we have also considered parameters to templates that are themselves templates, and this seems very useful. Below we first discuss type parameters about as they are presented in [28], and then the "formal template" kind (which were introduced in [38]).

Type Parameters.

As an example of type parameters, consider a template that implements a kind of list library, where each list will have a head object and a number of elements of parameter type `E`. Each element of the list is represented by an object of the internal class `AuxElem` that has a reference to the real list element of type `E`. This means that a given `E`-object can reside in any number of lists (and more than once in the same list). Type parameters can be constrained, and we include this in the example by assuming that each object of class `E` should be able to keep track of how many times it currently occurs in some list. For this purpose we require that the element class has two methods: `void incNo()` and `void decNo()` will increase or decrease the value of an internal counter, respectively, and these will be called at appropriate points in the methods `insertAsLast` and `removeFirst`, as seen in the example below:

```
template ListsOf<E implements {void incNo(); void decNo();}> {
   class List {
      AuxElem first, last;
      void insertAsLast(E e) { e.incNo(); ... }
      E removeFirst() { first.decNo; ... }
   }
   class AuxElem {
      AuxElem next;
      E e; // Reference to the real element
} }
```

We have here constrained `E` by an "anonymous interface" given as part of the parameter specification, so that an actual parameter for `E` has to implement the given methods. One can also constrain a type parameter by listing a number of ordinary interfaces that an actual parameter for `E` must implement, or by a class of which an actual class for `E` must be a subclass. If we as constraint use a class `C` defined in an ordinary package, this class `C` will be considered the same in all instantiations of this template and of other templates using the same bound. Obviously, inside the template we are allowed to use any property of a type parameter that follows from the constraints in a type-safe manner.

If we are to use a type parameter `T` with a bound `B` to create objects of `T`, then the following restrictions apply:

- `T` must be concrete.
- `T` must provide the same constructors as `B`, and `B` must thus be a class. Alternatively, one could envision a scheme similar to that of C#, where required constructors are specified explicitly as part of the parameter's bound. In the latter case, `B` could well be an interface (even an anonymous one).

In some cases it can be useful to use a type parameter `T` with a bound `B` as superclass for a class inside the template. In order to allow this, one must provide restrictions on

`T` so that `T` is not more restrictive with regard to what its subclasses can contain than `B` itself is. This implies the following restrictions:

- If `B` is concrete, then `T` must also be concrete.
- `T` must provide the same constructors as `B`.
- `B` must thus be a class.
- `T` cannot use covariant return types in overrides of methods from `B`.
- `T` cannot introduce `final` overrides of methods from `B`.

These restrictions should ideally be reflected by the bound `B`, but we currently have no specific syntax for this.

We do not allow template type parameters to also cover basic types (`int`, `byte`, etc.), e.g. like in C#. However, by introducing similar mechanisms for generics as in C#, this could also be allowed in PT, though with an additional restriction that parameters that are value types (primitives, structs, or enums) cannot be inherited from.

Templates with Template Parameters.

Letting templates abstract over regular type parameters provides a certain degree of flexibility that seems useful in many cases. However, a natural generalization of that mechanism would be to allow templates to abstract over templates, through the use of formal template parameters.

To be able to use templates as parameters in a meaningful way, we need a way to define bounds for them. Using a template `U` as a bound, a possibility would be to say that any template that instantiates `U` could be used in its place as an actual template parameter. However, relying on internal instantiations within a template would appear to break the principle of encapsulation, and we thus propose that a template can explicitly declare its instantiation of another template outside the template body, as shown below. We will refer to such instantiations as *explicit instantiations*, and correspondingly say that the clause specifying such an instantiation is an *explicit instantiation clause*. To illustrate this, we return to the `Vehicles` example from previous sections. Below, we see how a template `WeightVehicles` has an explicit instantiation of the `Vehicles` template:

```
template WeightVehicles inst Vehicles {
   class Vehicle adds {
      int weight;
      void display(){ tsuper.display(); out(weight); } }
   class Truck adds { ... }
   class Car adds { ... }
}
```

Note that `WeightVehicles` might make additions to the classes from `Vehicles` in the normal manner. A design decision here is whether one should be allowed to change the names of the elements from an explicit instantiation. This can probably be useful in some cases, but it will also make the structure of the program more difficult to follow. Thus, we assume for the rest of the paper that this is not allowed.

We can now write the following parameterized template, utilizing the `Vehicles` template as bound:

```
template RentalVehicles <template V inst Vehicles> {
   /* The rest is the same as in Section 3.2 */
}
```

The `RentalVehicles` template can subsequently be instantiated with an actual parameter that is `Vehicles` or a template that has an explicit instantiation clause that (transitively) includes the `Vehicles` template, such as e.g. `WeightVehicles` above.

```
package Program {
   inst RentalVehicles<WeightVehicles>;
   class Vehicle adds { ... }
   class Truck adds { ... }
   class Car adds { ... }
}
```

The instantiation of `RentalVehicles` will now instantiate `WeightVehicles`, which will instantiate `Vehicles`. The classes from `Vehicles` will form the base classes, `WeightVehicles` will then add its `adds`-clauses to these and then the package `Program` will make its additions through its `adds`-clauses, as shown above. Thus, calls in the `adds`-clauses of `Program` using `tsuper` will go to the (instance of the) `RentalVehicles` template, calls to `tsuper` there will go to `WeightVehicles` and calls to `tsuper` there will go to `Vehicles`. Note that the `tsuper` calls happen in exactly the same order as if `RentalVehicles` would have a normal (non-parameterized) instantiation of `WeightVehicles`, `WeightVehicles` would have a normal (non-explicit) instantiation of `Vehicles`, and `Program` would have a normal instantiation of `RentalVehicles`. Thus, the lookup algorithm for `tsuper` is still the same as the one presented in Section 12.3.2.

A template with an explicit instantiation clause can also be parameterized, and an important detail here is that the template that is explicitly instantiated can depend upon, or be one of, the template parameters, e.g. as shown below, where `U` is a formal parameter name, while `T` and `V` are actual templates:

```
template T <template U inst V> inst U { ... }
```

Interestingly enough, the construct above can be used to combine different variations of a shared base template. This can be used to solve the "expression problem" [20, 41] in a way that allows one to choose and combine different variations of a base

version of expressions as needed, and in the order one wants them. The expression problem is an example showing the limitations of single inheritance. If one has a set of expressions (implemented as subclasses of a common class, like `Exp` below) and a set of operations (implemented as virtual/abstract methods of that class) one may easily add new kinds of expressions by writing new subclasses (of e.g `Exp`), but adding new operations (virtual methods) requires changes to the existing classes. A technique that allows one to easily add new operations is the Visitor Pattern in [23], but that requires changes to the existing code to add new kinds of expressions. A solution to the expression problem allows one to add both new kinds of expressions (subclasses) and new operations (methods) without changing the existing code.

So in PT, the base template for expressions can e.g. look like this:

```
template Expressions {
   abstract class Exp { }
   class Plus extends Exp { Exp left, right; }
   class Num extends Exp { int value; }
}
```

Different variations of this template may add fields and methods to the classes, override methods, and add new classes.

Below are three examples of such variations of `Expressions`, written as templates that explicitly instantiate their parameter, in order to prepare them for subsequent composition. The first template adds a method to print the expression, the second one adds a method to calculate the value of the expression, and the third adds a new kind of expression node.

```
template PrintExpressions <template E inst Expressions> inst E {
   class Exp adds { abstract void print(); } // abstract
   class Plus adds {                         // extends Exp
      void print() { out("("); left.print(); out("+");
                     right.print(); out(")")}
   }
   class Num adds { // extends Exp
      void print(){ out(value);}
   }
}

template ValueExpressions <template E inst Expressions> inst E {
   class Exp adds { abstract int value();} // abstract
   class Plus adds {                       // extends Exp
      int value() { return left.value() + right.value(); }
   }
```

```
   class Num adds { // extends Exp
      int value(){return value;}
   }
}

template MultExpressions <template E inst Expressions> inst E {
   class Mult extends Exp { }
}
```

Now, we might want a version of expression that has all three of these variations, and the solution is to write this as follows:

```
package CombinedExpressions {
   inst MultExpressions<ValueExpressions<
      PrintExpressions<Expressions>>>;
   class Exp adds { }          // abstract
   class Plus adds { }         // extends Exp
   class Num adds { }          // extends Exp
   class Mult adds {           // extends Exp
      void print() { out("("); left.print(); out("*");
                     right.print(); out(")")}
      int value() { return left.value() * right.value(); }
   }
}
```

The code above works because all of the templates can take the place of the template `Expressions` as a template parameter. Furthermore, since they can all be instantiated with a template that explicitly instantiates `Expressions`, they can be combined in any order as parameters to each other and we can choose only the ones that are needed. The choice of order in such cases is usually not very important, but it defines in what order the `adds`-clauses are added and which method definition is reached using `tsuper`-calls from the different `adds`-clauses. Note how the original template `Expressions` is itself used as the parameter to the template `PrintExpressions` to form the base that the other templates successively add to. Note also that the class `Mult` has neither the `print` nor `value` method when originally defined in `MultExpressions`. Those methods are required in the program as `Mult` is a subclass of the abstract class `Exp`. The two required methods can conveniently be implemented in the `adds`-clause of `Mult` in the package `CombinedExpressions`.

There are a couple of complications that arise with template parameters that are not covered satisfactorily by bounds specifications. For templates with explicit instantiations (that can thus subsequently be used as actual template parameters), we have to introduce the following additional restrictions:

- additions to classes from an explicit instantiation cannot introduce method overrides with a covariant return type,

- method overloads cannot be introduced in additions to classes from an explicit instantiation.

Instance Parameters.

In this section, we have considered templates as parameters to templates. However, our research group has also been investigating the idea that *instances* of templates may be used as parameters to other templates. This opens up many possibilities, e.g. for dealing with shared template instances, and diamond-like structures of instances, while retaining most of the capabilities of the parameter mechanism described above. We leave a deeper exploration of this subject for future work.

12.3.8 Access Modifiers in PT

We have so far said nothing about access modifiers for PT, but they are obviously as important in PT as in any modern language, and perhaps even more so in PT since it is a system targeted directly at modularization of programs. When incorporating PT into a new language, there are at least two important questions with regard to access modifiers that need to be resolved. One is how to best adapt the access modifier system of the underlying language to the PT extension (or vice versa), and the other is to find the new interactions that turn up with PT that also should be regulated by some type of access protection. We will discuss these questions below, with the assumption that the underlying language is Java.

We first state the fact that *templates are effectively public*, just like packages in Java. That is, there are no access modifiers that are applicable to the template definitions themselves; access modifiers are applied only to the elements within a template, i.e. classes and interfaces, and their respective attributes.

Java Access Modifiers Used in Package Templates.

The Java access modifiers are: private, default (none specified, which means internal to the declaring package, also referred to as "package private"), protected and public. In a Java package, classes and interfaces can have either the default accessibility, or be public. It makes sense to allow the same modifiers for class or interface definitions in templates, and thus default means that the class or interface is accessible only within the declaring template, and e.g. not to templates or packages that instantiates this template. The default accessibility for definitions inside templates can as such be referred to as "template private". Correspondingly, a public template class will be accessible from everywhere within the declaring template, and also from everywhere within an instantiating template or package.

The same scheme that is described for classes in the previous paragraph can also be applied to the modifiers private, default (package private), and public when used for attributes in classes or interfaces. (Methods in Java interfaces are implicitly public, and the same should be true for template interfaces.) However, the modifier protected turns out to have some interesting aspects when applied to PT, and we will get back

to this modifier shortly. Another interesting issue concerns the modifier public, and the question of whether public elements in a template will always also be public in the template/package that instantiates them. We will get back to that later in this section.

The `protected` Modifier, and Potential New Access Modifiers.

In Java a `protected` attribute of a class is accessible from anywhere in the enclosing package, and also from subclasses of that class defined outside the package. However, in PT a new element similar to subclasses has emerged, namely the addition class. We may then ask whether a `protected` attribute should also automatically be accessible from an addition class, or whether it sometimes is convenient to say that an attribute is accessible from a subclass, but not from an addition class (or the other way around). From the programming and sketching we have done, it often seems natural to consider the `adds`-part as being closer to the template class than a subclass is. This should indicate that we need a modifier that expresses that the attribute is accessible from an addition class, but not from a subclass. However, for symmetry, we should then probably also have the opposite one, and we could e.g. call them `aprotected` and `eprotected` ('a' for *adds* and 'e' for *extends*). We could then let the traditional `protected` modifier mean that the attribute is accessible from both dimensions.

One could also discuss whether we need a modifier that says that an attribute is accessible from an addition class, but *not* from a subclass nor from the rest of the template. In C#, protected has this more restricted meaning for subclasses (i.e. it does not, as opposed to Java, provide access to the entire "package"). However, we feel that this is more a discussion about Java versus other languages than about PT in itself, and we will thus not pursue that question in this paper.

Access Modifiers for Instantiations.

In the previous subsections, we discussed the use of access modifiers within templates. However, when a template is instantiated in a scope (package or template) there is also a need to regulate the accessibility outside that scope of the definitions received from the instantiated template. The most natural place to control this is in, or somehow connected to, the `inst` statement. We therefore propose the concepts of `private` and `public` instantiations. In a `private` instantiation everything that was made accessible to the instantiating scope, will in this scope be considered "package private" (or "template private"). This might typically be used in situations where an instantiated template is used for the internal implementation of some functionality, and where one does not want to expose such implementation details to subsequent users of this functionality.

On the other hand, in a `public` instantiation everything that was made accessible to the instantiating package/template, will get the same accessibility in this scope. This could typically be used when the instantiated template is some type of framework, and one in the instantiated scope only wants to add some final definitions. Note that it should also be possible to have modifiers on the addition classes in the instantiation

scope, and a reasonable rule is that if a modifier occurs it overrides the one from the template.

We think that the accessibility system described above is promising, but based on our limited programming experience with PT in a large scale setting, it seems to be too early to conclude. Thus, as we gain experience and hopefully get external input, we hope to revisit this topic as part of our future work.

12.3.9 Implementation

A mechanism like PT can basically be implemented in two different ways. One is a so-called heterogeneous implementation, where each instantiation of a template results in the insertion of the relevant program text (or e.g. byte code) into the instantiating package or template. The resulting package can then be compiled as a whole. This resembles how templates are implemented in C++. A potential problem with this strategy is that we might end up with a lot of code at runtime (a problem often referred to as "code bloat"). However, the problems associated with this are probably less pronounced now compared to some years ago, as the amount of available memory has been growing steeply in recent years.

The implementation we are currently working on is of the heterogenous type, and it is built upon the JastAdd system [16, 17]. In addition to its extensibility, JastAdd also provides a good compiler for traditional Java "for free".

The source code for the prototype compiler can be downloaded from the following url: `http://swat.project.ifi.uio.no/software/`. This compiler currently implements the main concepts of this paper, but not some of the more recent additions to the PT mechanism such as the `tabstract` modifier and template parameters.

The output from the prototype compiler is plain Java source code, which can then subsequently be compiled using the standard `javac` compiler. When the compiler generates the Java code, it mangles overridden method names from template classes, and subsequently transforms `tsuper` calls to ordinary calls to the mangled names. Template class constructors are implemented in a similar manner.

Since all substitutions and additions in PT are done semantically, one might also try to make a homogenous compiler, which means that only one version of the code for a template is produced and stored during execution, and tables and other mechanisms are used to keep track of the different instantiations and their adaptions to the stored code. The problem here is the speed of the execution, e.g. since merging will lead to many of the same problems as multiple inheritance does. However, we have some ideas on how this can be done in an efficient manner, and hope to be able to test them out in the near future.

12.4 Related Work

The trait mechanism [37] approaches composition from the angle that the unit of composition is a trait and that a trait or a class can be composed of traits. A trait is a

stateless[1] collection of methods. The methods of the traits become methods of the new trait or class that is composed of traits. In addition to the methods that contribute to the composed trait or class, a trait may also specify required methods, i.e. methods that it requires the composed trait or class to have, either from other traits or from the class definition itself. The composition of traits is said to be *flattened*. This means that (1) the trait ordering in each composition is irrelevant, and (2) that a class composed from traits is semantically equal to a class in which all the methods are defined directly in the class. Traits were originally developed for the dynamic language Squeak, and support method aliasing and exclusion upon composition. A statically typed version also exists [33].

PT is similar to traits in the following way: When classes are merged, all the methods from the merged classes are included in the resulting class. PT does not have a mechanism like required methods, and while the `tabstract` modifier might seem similar, there are quite a few differences. When a class with a `tabstract` method is merged with other classes, the `tabstract` method must be implemented in the addition class (or remain `tabstract`). If one of the other classes has a method that matches the signature of the `tabstract` method it will only be a name collision that has to be resolved. Another difference between traits and PT is that in PT composition is performed at the level of a package (template) and thus across more than one class, while trait composition applies to single traits or classes.

Mixins [9] are similar in scope to traits in that they target the reuse of smaller units that are composed into a class. Mixins also define provided and required functionality, and the main difference between them and traits is the method of composition. Mixins traditionally rely on inheritance, by defining a subclass with an as-of-yet undefined (virtual) superclass, and the result is that mixins are linearly composed. Mixins allow super calls to methods in that superclass. The actual superclass is specified when the mixin class is used in a program.

Calling methods in an as-of-yet undefined (virtual) superclass can also be achieved in PT by using templates as template parameters. This is what is done in the example with the expressions in section 12.3.7 where the different additions to the same class (like `Exp`) are layered; one adding to and overriding another in the order specified in the `inst` statement in the program and where a `tsuper` call would call the overridden method in the class from the "previous" template in the sequence. As with traits, another difference is that in PT composition is performed at the level of a package (template) and thus across more than one class at once.

Virtual classes were introduced in the BETA language [30, 31], and have subsequently formed the basis for a number of languages, such as e.g. gbeta [18], J& [35] (pronounced "jet"), and Caesar [1] (the latter also contains AOP features, see below). The main idea is that a class encloses one or more inner classes that are virtual in the sense that they can be overridden in subclasses of the enclosing class. Except for BETA, the mechanisms also allow some kind of merge (or multiple inheritance) of enclosing classes, with merging between inner classes following from that.

[1] Traits were originally defined to be stateless, although a more recent paper [7] has shown how a stateful variant may be designed and formalized.

Mechanisms based upon virtual classes provide some of the same flexibility for unanticipated adaption as PT's addition classes. The main difference is that PT does not rely on nested classes, there will thus be no objects of an enclosing class and no types dependent on such runtime objects. Also, there is no multiple inheritance in PT since combination of classes from different templates is handled with the merge construct. Since the different packages (or templates) that instantiate a template are not subtypes of the template they instantiate, this gives PT some flexibility in what one can do at merge time, but it does not allow full family polymorphism [19].

Lightweight family polymorphism [36] makes very much the same trade-off that PT does in that it represents families by classes instead of objects, whereas PT represents them by templates and packages. Thus, according to the authors, one gets a simpler type system but with some restrictions. They do not discuss merging classes from different families which is one of the main features of PT. Variant path types [26], which is an extension of Lightweight family polymorphism also introduces inheritance between classes within a family. The type system is still somewhat simpler than those of languages like gbeta and J&, but also still not as expressive. Variant path types also makes much the same trade-offs as PT, but with its exact types and inexact types and partially inexact types with their inexact qualification and exact qualification one could argue that PT is even simpler. They do allow methods that work uniformly over different families.

Classboxes [6] were designed to allow unanticipated and unintrusive changes, like adding fields and methods to existing class hierarchies and make the expanded classes available to new clients without affecting existing clients. Classboxes do not have a mechanism for merging classes like PT has, nor does it have any form of multiple inheritance, but it is similar to PT in that different additions to a base hierarchy can be combined in a layered fashion. Method overriding and lookup are very similar, with separate keywords for `super` and `tsuper` (the latter called `original` in Classboxes).

MultiJava [11] also allows methods to be added to existing classes and also allows such added methods to be overridden by methods added to subclasses, but it does not allow methods in the existing classes to be overridden by the so-called external methods. MultiJava also has symmetric multiple dispatch which neither Classboxes nor PT has.

Expanders [42] is another mechanism for adding methods and fields to existing classes that only affect a well defined scope. With Expanders methods and fields can be added to interfaces and interfaces can also be added to classes. The mechanism has modular type safety and existing objects can be updated, even with new state. Methods in expanders can override methods in other expanders, but not methods in the expanded classes. Like the other mentioned mechanisms for expanding classes it does not have a merge construct.

Difference-Based Modules in MixJuice [25] are similar to PT in that the modules define or adapt more than one class, that modules can extend other modules, and that they can be combined to form new modules. Like PT, MixJuice also makes a clear distinction between subclassing and adapting a class and uses the keywords `super` and `original` where PT uses `super` and `tsuper`. It does not have a renaming or

merge construct for adapting classes and combining classes that were independently written.

DeepFJig [14] uses nested static classes as modules and, like PT, it is designed for flexibility in combining modules. When (outer) classes are combined, inner classes are merged recursively when they have the same name. More than one abstract method or field with the same name is joined into one while a non-abstract one replaces or implements the abstract one. DeepFJig has a series of composition operators so that renaming, hiding and overriding can be done in much the same way as in PT. However, subtyping within a module (outer class) is different in that in DeepFJig one only declares a class to implement another and thus all elements must be (re)declared in that class, which is unlike regular inheritance which one can use within a package template, where elements are inherited from the supertype. DeepFJig does not have the separate `super` and `tsuper` calls that PT has and composition operators must be used to rename methods to achieve the same. Unlike PT, there does not seem to be any way to refer to a shared base when writing and combining different extensions in DeepFJig, like in the solution to the expression problem in [14]. PT does not allow classes, methods or fields to be removed or hidden from templates.

Newspeak [10] is a dynamically typed class based language descending from Smalltalk and Self, with no global state or namespace. All classes, including superclasses, are virtual and they can be nested arbitrarily. Newspeak is similar to PT in that there can be more than one version of a module (class in Newspeak and template in PT) at runtime and that the "actual imports" (instance parameters in Newspeak and templates as template parameters in PT) are decided by the client. Unlike Newspeak, PT has a global namespace and, like BETA, PT is based on further binding and not on completely replacing a class like replacing a virtual method. Furthermore, in Newspeak, module instances are per instance (object), while in PT they are per class or package.

Ada originally (in 1983, [29]) had no mechanisms supporting object-orientation, but it had a mechanism called generic packages with some of the same aims as package templates, in that generic packages can contain type definitions and that each instantiation of a generic package gives rise to a new set of these types.

In Ada 95 [5] a mechanism for object-orientation was introduced (further elaborated in Ada 2005). As far as the authors understand it, there is nothing similar to virtual classes (at compile-time or at runtime) in the language, and the mechanisms for adapting a package to its use are not very advanced.

Aspect-oriented programming [1, 12, 27] (AOP) is an approach to separation of concerns and code reuse where an important notion is that of crosscutting concerns, i.e. concerns that are not easily captured in just one class of an OO application. In [22] the authors state that AOP in essence is *"quantification and obliviousness"*, indicating that, according to their view, AOP involves quantification of program locations to affect and that code should be unaware of the aspect code that will affect it. In that sense, aspects can be seen as a special case of meta-level entities that examine and manipulate programs. The pointcut-mechanism has received some criticism for its fragility with respect to changes in the involved classes, known as the *fragile pointcut problem* [39].

Package templates are oblivious to both their potential usage and changes made to

them by the `inst` statement or corresponding addition classes. The `inst` statement can also be seen as a limited way of quantifying program elements, by naming templates to instantiate and by specifying class merges to obtain the desired end result. So-called intertype declarations play an important role in many AOP solutions. In PT, such additions can be made to classes either through merging, or through the use of addition classes.

However, while PT as presented in this paper can solve a certain category of AOP problems, there are quite a few things that seem to better be approached through a mechanism that includes some notion of pointcut and advice. Thus, we have also performed some experiments with adding more explicit AOP support to PT, see e.g. [4]. In that paper, we strive to find a middle-way between the more restricted `inst` mechanism and some of the power of expression inherent to e.g. AspectJ, and show how a restricted version of pointcut and advice definitions can be incorporated into template classes and how this can be utilized to create a reusable implementation of the Observer design pattern [23, 24, 32]). The gist of the paper [4] lies in the fact that reusable entities like design patterns can be merged with other template classes in order to add the functionality of the pattern directly to them, and that abstract pointcut specifications can be concretized in addition classes, utilizing both definitions from the design pattern and from the code with which the pattern is merged.

12.5 Concluding Remarks

In the introduction we stated that we want PT to represent a compromise between (1) simplicity of the type system, compared with type systems for approaches built on virtual classes, and (2) expressiveness, especially for large-scale programming. This overall goal has thus been used as a guide throughout the design process in face of diverging design considerations.

Concerning (1), with a simple type system we mean a type system that is mainly easy to understand and use, but also relatively easy to implement. Below we list a few points that we think support the view that the type system of PT generally is simpler than those of systems built on virtual classes.

- The main thing that complicates type systems for approaches based upon virtual classes is that the types corresponding to inner, virtual classes depend on the object of the enclosing class. There may be several objects of this enclosing class, each of these will have the same type (and may therefore potentially be denoted by the same reference), while types according to the inner, virtual classes are different for different objects. Template classes of PT are not ordinary classes. Instantiation of a template is performed at compile-time, and it results in a set of ordinary classes, not in the scope of an enclosing object, but in the scope that instantiates it. Several instantiations of the same template will result in independent sets of classes.

- Instantiation of templates and especially merging of template classes can be described as program compositions. An implication of this is that the template

classes form constituent parts of the resulting ordinary classes, so that they do not exist as separate classes after the instantiation, and there will not be objects of the template classes themselves. This is generally not the case for virtual class systems, where a class and an extension of it will often both exist and even have the same name, so some mechanisms must be introduced to distinguish them.

Concerning (2), we have demonstrated that package templates solve some well–known problems, like e.g. the expression problem. With respect to large scale programming we still need some more evidence. Hopefully, for some of the more contested design issues, such as constructor call schemes or access modifiers for template classes, the right solution will become more evident as our experience with programming with the PT mechanism grows.

With respect to reuse it is worthwhile to compare with e.g. gbeta [18] and the paper on Family Polymorphism [19] by E. Ernst. Here it is claimed that for an approach to really support reuse, it must be possible to dynamically generate any number of specific class collections (which in his setting are objects of an outer class) from a general class collection. We agree that it is important to be able to generate multiple instantiations of the collections, but we think that for most purposes it is enough to generate them statically. Combined with the mechanism of adaptions, package templates support that specific collections suit specific needs. Approaches based on virtual classes do not (by nature of the underlying mechanism) support the flexibility of renamings and merges.

The added benefit of using package templates that can be instantiated and adapted multiple times during compilation will represent a significant step forward compared to e.g. ordinary packages, and we believe that the final step towards dynamic instantiations are important only in very special cases.

Acknowledgements

We would like to thank Eivind Gard Lund and Ivar Refsdal for their dedicated work on the PT compiler. Furthermore, we would also like to extend our gratitude to the anonymous reviewers for their insightful and valuable comments on an earlier version of this paper.

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30).

Bibliography

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] Eyvind W. Axelsen and Stein Krogdahl. Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 15–26, New York, NY, USA, 2009. ACM.

[3] Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller-Pedersen. Controlling dynamic module composition through an extensible meta-level API. In *DLS 2010: Proceedings of the 6th symposium on Dynamic languages*, New York, NY, USA, 2010. ACM.

[4] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 37–42, New York, NY, USA, 2009. ACM.

[5] John Barnes. *Programming in Ada95*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[6] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: controlling the scope of change in Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[7] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[8] Gilad Bracha. Generics in the Java programming language. Technical report, Sun Microsystems, Santa Clara, CA, July 2004. `java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[9] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *OOPSLA/ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010.

[11] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28:517–575, May 2006.

[12] Adrian Colyer. AspectJ. In *Aspect-Oriented Software Development*, pages 123–143. Addison-Wesley, 2005.

[13] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[14] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig - modular composition of nested classes. In *2010 International Workshop on Foundations of Object-Oriented Languages (FOOL '10)*, 2010.

[15] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical Report Publication No. S-22 (Revised edition of publication S-2), Norwegian Computing Center, October 1970.

[16] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[17] Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[18] Erik Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[19] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[20] Erik Ernst. The expression problem, scandinavian style. In Philippe Lahire, Gabriela Arévalo, Hernán Astudillo, Andrew P. Black, Erik Ernst, Marianne Huchard, Markku Sakkinen, and Petko Valtchev, editors, *MASPEGHI 2004*, 2004.

[21] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM.

[22] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Rober E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *AOSD*, pages 21–31. Addison-Wesley, 2005.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[24] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173, 2002.

[25] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 62–88, London, UK, 2002. Springer-Verlag.

[26] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 113–132, New York, NY, USA, 2007. ACM.

[27] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *LNCS*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[28] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[29] Henry Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[30] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[31] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[32] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03*, pages 90–99, New York, 2003. ACM.

[33] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[34] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM.

[35] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[36] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *J. Funct. Program.*, 18:285–331, May 2008.

[37] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[38] Fredrik Sørensen, Eyvind W. Axelsen, and Stein Krogdahl. Reuse and combination with package templates. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 3:1–3:5, New York, NY, USA, 2010. ACM.

[39] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, Berlin, Germany, September 2004.

[40] Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, 1989.

[41] Mads Torgersen. The expression problem revisited. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143. Springer, 2004.

[42] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 37–56, New York, NY, USA, 2006. ACM.

Chapter 13

Paper VI: Adaptable Generic Programming with Package Templates and Required Types

Authors. Eyvind W. Axelsen and Stein Krogdahl

Publication. Proceedings of the 11th annual international conference on Aspect-oriented Software Development, 2012.

Abstract. The aim of this work is to provide better support for adaption and refinement of generic code. This type of flexibility is desirable in order to fully reap the potential of generic programming. Our proposal for an improved mechanism is an extension to the previously published *Package Templates* (PT) mechanism, which is designed for development of reusable modules that can be adapted to their specific purpose when used in a program. The PT mechanism relies on compile-time specialization, and supports separate type checking and type-safe composition of modules. The extension to PT presented here is called *required types*, and can be seen as an enhanced form of type parameters, allowing them the same flexibility as other elements of the PT mechanism. We implement a subset of the *Boost Graph Library* in order to exemplify, validate, and compare our approach to other options.

Categories and Subject Descriptors. D.2.13 **[Software Engineering]**: Reusable Software; D.3.3 **[Programming Languages]**: Language Constructs and Features — *Modules, packages*

General Terms. Languages, Design

Keywords. Generic Programming, Reuse, Templates

13.1 Introduction

When developing libraries or other software components meant for widespread reuse, it is vital to minimize assumptions on the client code. On the other hand, it is equally important to be able to express a sufficient set of requirements for the clients of the library so that it can be written in a type-safe manner that yields efficient code. Furthermore, it is important that a client can refine and adapt the library to the problem at hand.

Many languages, such as e.g. Java, C# , C++, Scala, and Haskell, support constructs for *generic programming* in order to better facilitate the development of reusable libraries. The degree to which each language supports such constructs varies, and an excellent overview that compares several languages with respect to support for generic programming can be found in [10].

There are several definitions of what generic programming actually entails (or should entail), but perhaps a more fruitful angle is to consider what it is that we are trying to achieve with such mechanisms. In that respect, Jazayeri et. al [14, page 2] state the following:

> "The goal of generic programming is to express algorithms and data structures in a *broadly adaptable*, interoperable form that allows their direct use in software construction" [emphasis ours].

We agree, to a large extent, with this quote, even if it may be deemed a bit wide in scope. The adaptability part of the goal is in our opinion very important, and in this paper we will describe a mechanism that we think in many cases can represent both an improvement and a simplification with respect to adaptable generic programming compared to contemporary approaches.

The mechanism is an extension of the Package Template (PT) mechanism [3, 15]. We will in the following refer to the previously published variant as *basic PT*, or just PT when the variant is obvious from the context. Basic PT allows type safe renaming, merging, and refinement in the form of static additions and overrides that are orthogonal to ordinary inheritance. It thus differs from typical virtual class-based [16] mechanisms in that composition and refinement (beyond ordinary OO constructs) is reified at compile-time only, yielding a simpler type system.

Seeking to also attain the goal presented above for *generic*, potentially *heavily parameterized*, libraries, we incorporate a notion of *required type specifications* in the PT mechanism, and we label this variant PT*r*. The approach is inspired by suggestions to use virtual types as an alternative approach to generic parameterization in Java [26], and enables utilization of basic PT's inherent capabilities for adaption also for generic concepts and constraints. PT*r* supports multi-type concepts, associated types, and nominal and structural generic bounds. Retroactive modeling and adaption through renaming, merging and additions are thus also supported, without sacrificing type safety, performance, or dynamic dispatch.

To demonstrate and validate our approach, we have implemented a small yet non-trivial subset of the Boost Graph Library [23], which employs a rather advanced usage

of generics. The subset is the same as that described and implemented by [10], and we will compare and contrast the implementation made possible with *PTr* with those of [10]. The implementation and a prototype compiler can be downloaded from `http://swat.project.ifi.uio.no/software`.

The main contribution of this paper is thus to present PT*r* as an approach to creating flexible generic libraries, and to demonstrate through a non-trivial example its benefits and tradeoffs.

The rest of this paper is organized as follows: Section 13.2 presents necessary background material and introduces basic PT (13.2.1), and the Generic Graph Library (13.2.2) through a discussion on criteria for generic constructs in general and the goals of our mechanism in particular. Sections 13.3 and 13.4 contain a description of the proposed addition of required types to PT, and an overview of how PT*r* fulfills most of the criteria presented in Section 13.2.2, respectively. Related work is treated in Section 13.5, and Section 13.6 concludes this paper.

13.2 Background

13.2.1 A Brief Overview of the Basic PT Mechanism

In this section we give a general overview of the basic PT mechanism. The concepts of the mechanism are not in themselves tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax. The interested reader is referred to [3, 15] for a more thorough exposition.

A package template looks much like a regular Java package, but we will use a syntax where curly braces enclose the contents of both templates and regular packages, e.g.:

```
template T<R> { // R is not discussed here, see Sec. 3
  class A { ... }
  class B extends A { ... } }
```

In contrast to for instance templates in C++, package templates can be type checked independently of their potential usage(s).

A template is instantiated at compile time with an `inst` statement. Such an instantiation will create a local copy of the template classes, potentially with specified modifications, within the instantiating package or template. An example of this is shown below:

```
package P {
  inst T with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D extends C since B extends A
}
```

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `P`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. "`inst T`". However, modifications can also be made to the template classes upon instantiation, such as:

- Elements of the template may be renamed. This is done in the `with`-clause of the `inst`-statement, and is only shown for class names above (`A` is renamed to `C` and `B` is renamed to `D`). For renaming of class attributes another arrow is used (`->`). Note that all renaming in PT is done based on the name bindings from the semantic analysis.
- In each instantiation the classes in the template may be given additions: fields and methods may be added and virtual methods may be overridden. This is done in `adds`-clauses as shown for `C` and `D`.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is updated to instead refer to the corresponding names of the addition classes (`C` and `D`) at the time of instantiation. Any sub/super-type relations within the template are preserved in the package where it is instantiated. Note that templates can also be instantiated in other templates.

Classes from different template instantiations may be *merged* to form one new class. Syntactically, merging is obtained by renaming classes from two or more template instantiations to the same name, and they thereby end up as one class. The new class gets all the attributes of the instantiated classes, together with the attributes of the common addition class. Consider the simple example below:

```
template T { class A { int i; A m1(A a) { ... } } }
template U {
  abstract class B { int j; abstract B m2(B b); }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;
class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the abstract method `m2` from `B` is implemented in the `adds` clause, and furthermore how both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`.

Merging classes in this manner might obviously lead to name clashes; such conflicts must be resolved through renaming.

13.2.2 The Generic Graph Library and Evaluation of Generic Support

For the purpose of demonstrating and validating the generic programming constructs added to PT in this paper, we have implemented a small yet non-trivial subset of the Boost Graph Library (BGL) [23], revolving around a set of algorithms using variants of breadth-first search, including Prim's minimum spanning tree, Dijkstra's shortest paths, Johnson's shortest paths, and Bellman & Ford's shortest paths algorithms. The implemented subset is the same as that of [10].

In the implementations from [10], emphasis is put on expressing minimal requirements for each algorithm. These have internal (acyclic) dependencies, e.g. Johnson's algorithm depends on Dijkstra's algorithm.

The graph itself is represented in terms of *concepts*. The term concept is in [10] used to mean a set of requirements consisting of required operations (methods) and data type constraints. A type (or a set of types) is said to *model* a concept if it (they) fulfill(s) these requirements. In a Java-like language, concepts are typically realized as interfaces[1], and classes implementing such an interface thus model the corresponding concept. The following main concepts of the graph library are in Java implemented as interfaces:

- `VertexListGraph`: provides an iterator yielding all vertices in the graph in an unspecified order.
- `EdgeListGraph`: provides an iterator yielding all edges in the graph in an unspecified order.
- `IncidenceGraph`: provides an iterator yielding the directed edges going out of a given vertex.

These concepts are used by the algorithms to express constraints on their input parameters. For convenience, concepts that are combinations of the aforementioned ones are introduced, e.g. the `VertexListAndIncidenceGraph` concept which is an interface that extends both the interfaces representing the `VertexListGraph` concept and the `IncidenceGraph` concept. Additionally, the different algorithms require various data structures for coloring, ordering, etc., realized as e.g. property maps, queues, etc. These structures are supplied explicitly as parameters to each algorithm.

Several languages were studied in [10], and subsequently evaluated based on their support for generic programming constructs. Table 13.1 shows an overview of the rating for C++ and Java from that paper, plus an additional column for Scala, the latter taken from [21]. The different categories in the table are described in Table 13.2A, taken from [10]. An extended evaluation, including PT*r*, will be presented in Section 13.4.

While the original study included several other languages as well, we focus on Java and C++, and in addition we include Scala. C++ is interesting because its generic capabilities rely heavily on its templating mechanism, and it is in this language that

[1] See [21] for an alternative approach based on the Concept pattern.

	C++	Java	Scala
Multi-type concepts	*	○	●
Multiple constraints	*	●	●
Associated type access	●	◐	●
Constraints on assoc. types	*	◐	●
Retroactive modeling	*	○	●
Type aliases	●	○	●
Separate compilation	○	●	●
Implicit argument deduction	●	●	●

Table 13.1: The table shows the level of support for generic programming constructs for C++, Java, and Scala. The table criteria and the support levels for the former two are taken directly from [10, page 147]. A black circle indicates full support, a half-filled circle indicates partial support and a white circle indicates poor or no support at all. For C++, a rating of '' means that the feature is not explicitly supported by the language, but the permissiveness of the language allows one to program as if it were supported, though sans compiler support.*

the Boost Graph Library has its native implementation. However, C++ templates differ drastically from the templates of PT, most notably in the sense that PT templates are declarationally complete semantic units that can be type checked independently of their usage. As can be seen from the table, while much can be achieved in C++ due to its flexibility with regard to template definition, we only get limited compiler support.[2] Java is obviously interesting because the lacking points for Java in the table is the situation that we wish to ameliorate with PT*r*, and PT is designed as an extension to Java-like languages. Scala is a rather advanced JVM language that also addresses many of the weaknesses of Java with regard to generic programming, and as such it is an interesting language with which to compare and contrast our mechanism.

A partial goal for this paper can thus be summarized as *to bring some of the flexibility of C++ generic template programming to Java with PTr, while retaining compiler support, static safety, and relative simplicity*.

However, if we look back to the goal from [14] presented in the introduction, we argue that the criteria from [10] do not adequately encompass requirements for writing algorithms and data structures that are *broadly adaptable*. For instance: How can a constraint for a generic parameter be adapted to match existing code? How can a constraint be refined by subsequent users? How can a concept be adapted to allow modeling by existing data structures, or vice versa?

In order to cover these usage scenarios, we introduce two new criteria related to adaptability, presented in Table 13.2B. Thus, another part of our goal with this paper is *to satisfy the adaptability criteria for generic programming*.

[2]Note, however, that Java-style generics can be emulated in C++, with compiler support, through use of for instance the Boost Library's `BOOST_STATIC_ASSERT`.

Scores for Scala. In [21], Oliviera et al. discuss the criteria from Table 13.2A in context of the Scala language. In their treatment, Scala receives the *full support* verdict on all points, as shown in Table 13.1. One could thus think that there is little room for making improvements with PT*r*. However, we have found that, at least with respect to an implementation of the generic graph library, Scala still leaves a few things to be desired with respect to these criteria. Furthermore, as we shall see, PT*r* takes quite a different approach to generics compared to Scala. Also note that [21] changes the scores for Java, but we have kept them in their original form from [10]. We refer to the discussion of the individual criteria in Section 13.4 for details.

13.3 Required Type Specifications in PT*r*

Basic PT [3, 15] allows templates to have generic type parameters in much the same way as ordinary Java classes can, e.g. as in

```
template T<R> { ... }
```

The parameters may be constrained, either through a nominal inheritance specification (akin to Java generics) or through a structural requirements specification, e.g.:

```
template T1<R extends Runnable> { ... }
template T2<R extends { void run(); }> { ... }
```

While this provides the ability to let the template classes be collectively parameterized, which in itself can be very useful, the approach has certain limitations. To begin with, the parameter `R` in the examples above does not naturally lend itself to the kind of modification that are allowed for basic PT classes, e.g. renaming of attributes, merging etc. Since the parameter is part of the template specification, it could be natural (or even necessary) to adapt the parameter specification along with other adaptions of the template. Furthermore, neither the name `R` nor the requirement it poses is *propagated* to other templates that instantiate `T`, `T1`, or `T2`. As we will see examples of below, and as was demonstrated in [10], this is a real issue for the implementation of larger libraries as it often leads to unnecessary code duplication. Also, as a consequence of the lack of propagation, there is no straightforward way to refine constraints without cumbersome and error-prone repetition of code.

With basic PT, class declarations in a package template can, as we have seen in Section 13.2.1, be adapted in several ways. It seems like a natural step forward to provide the same degree of flexibility for parameterization of templates. Thus, this can be seen as making the generic constraints of a template *first class entities* of the template, in the same way that classes and interfaces are. In the rest of this section, we will look at how PT*r* provides this feature.

Required types as first-class template declarations. The syntax for required types in PT*r* is summarized in Figure 13.1. A basic specification requiring an unconstrained type, i.e. any Java class or interface, to be supplied can be expressed as follows:

	Criterion	Definition
A)	Multi-type concepts	Multiple types can be simultaneously constrained.
	Multiple constraints	More than one constraint can be placed on a type parameter.
	Associated type access	Types can be mapped to other types within the context of a generic func.
	Constraints on associated types	Concepts may include constraints on associated types.
	Retroactive modeling	New modeling relationships can be added after a type has been defined.
	Type aliases	A mechanism for creating shorter names for types is provided.
	Separate compilation	Generic functions can be type checked and compiled independent of calls to them.
	Implicit argument deduction	The arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.
B)	Retroactive concept adaption	Concepts can be adapted after their initial definition. If the concept spans multiple types, a single adaption may affect several types.
	Retroactive constraint adaption	Constraints for generic parameters can be adapted and refined after their initial definition to better match existing code.

Table 13.2: A) The different criteria for evaluation of support for generic constructs in programming languages, taken directly from [10, page 147]. B) Additional criteria for evaluation of support for adaptable generic programming. We discuss these criteria in further detail in Section 13.4.

```
template T1 { required type R { } }
```

An actual parameter for `R` can be supplied to `T1` when instantiating the template, by utilizing the <= arrow. To e.g. supply `String` for `R` in `T1`, the following instantiation could be used (within another template or package, see separate paragraph on that below):

```
inst T1 with R <= String;
```

At this point we see an important difference between required types in PT*r* and type parameters as found in e.g. Java, Scala or C# with regard to their scope: Required types are at the same lexical level as class declarations, and can thus naturally constrain a *set* of classes. This is to some extent similar to abstract types and nested classes within an outer class in e.g. Scala, but it is important to note that required types in PT*r* (and package templates as a whole) is a compile-time construct only. Thus, after instantiation of the template in a package, there will be no inner abstract types (and thus no full family polymorphism [7] nor path-dependent types). This amounts to a simpler type system, and is as such comparable to the flattening property of traits [22].

A required type `R` can be constrained by both structural and nominal specifications; below we see an example of the former:

```
template T2 { required type R { void run(); } }
```

Given an instantiation that supplies an actual type for a required type `R` with a structural constraint, such as "`inst T2 with R <= Runnable`", the compiler will check that the supplied type structurally conforms to the specification given by `R`. Conformance entails that all the required methods (and constructors if they are present, see below) must have an *exact* match (save for parameter names) in the supplied type. Covariant or contravariant signatures are not allowed; allowing this would not be type safe (e.g. when merging). Thus, for required types with only methods, the conformance relation for required types is equivalent to the <# matching relation of the $\mathcal{LOOM}$ language [5].

If the signature check succeeds, the compiler will *replace* all occurrences of `R` (based on the semantic analysis) in the instantiated template with the supplied type; in the example instantiation in the paragraph above, references to the type `R` will be replaced by references to the type `java.lang.Runnable`. The actual declaration of the required type `R` will be removed. Thus, for every class in the template, a version specific to this instantiation, with the given parameterization, is created. Note that, as opposed to in e.g. Scala, there is never a need for runtime reflection when dealing with structural constraints, since the actual type always will be known at *compile-time*.

Bounds for required types can be specified nominally as well as structurally. Taking the example from above, we can express that `R` must be a nominal subtype of e.g. the `Runnable` interface:

```
template T3 { required type R extends Runnable { } }
```

required-spec	::=	`required` [<r-type>|<r-class>|<r-interface>]
r-type	::=	`type` <identifier> [`adds`] [<extends-clause>] `{` <r-type-body>* `}`
r-interface	::=	`interface` <identifier> [`adds`] [<extends-clause>] `{` <r-type-body>* `}`
r-class	::=	`class` <identifier> [`adds`] [<implements-clause>] [<extends-clause>] `{` <r-class-body>* `}`
r-type-body	::=	<method-signature>
r-class-body	::=	<constructor-signature> | <field-signature> | <method-signature>

Figure 13.1: *Syntax for required types. Non-terminals are written within <angled brackets>, and optional symbols are delimited by [square brackets]. A vertical line* (|) *signifies alternatives, and a star* (*) *signifies zero or more repetitions of a symbol. Terminal symbols are written with a* `monospace font`. *Productions left out (for the sake of brevity), such as e.g. the* extends-clause*, are to be understood as syntactically equal to their pure Java equivalents.*

Thus, when supplying an actual type `A` for `R` in an instantiation of `T3`, it must explicitly implement or extend the `Runnable` interface (or `A` might be the `Runnable` interface itself).

Nominal and structural subtyping can also be mixed in a declaration of a required type. To demand an explicit, nominal, implementation of `Runnable`, and furthermore that a method `stop` must be present, we can easily express this as follows:

```
template T4 {
   required type R extends Runnable { void stop(); }
}
```

Classes and interfaces. Java generics do not allow the use of primitive types (though Scala does), and we do not in this work intend to lift that restriction. However, it is still important in some cases to be able to explicitly constrain a required type to be either an interface or a class.[3]

As can be seen from the syntactical overview in Figure 13.1, such constraints can be imposed by declaring a required type explicitly as either a `required interface` or a `required class`. Declaring requirements in this way puts further constraints on the required types; e.g. the ability to have constructors and fields are only available to required classes.

Thus, the term *required type* is overloaded in this paper, and is used both in the inclusive sense to refer to the syntactical and semantical constructs of required *types*, required *classes* and required *interfaces*, and, on the other hand, in the narrow sense to *only* refer to required *types*. When this distinction is important, we will be explicit about this; otherwise, the inclusive sense is implied.

[3] An example of a similar construct in a mainstream OO language is the `where R : class` constraints of C#.

Instantiation and concretization As mentioned above, upon instantiation of a template, the programmer may choose to supply an actual concrete type for a required type `R` that satisfies the constraints of `R`. We will refer to this as a *concretization* of the required type.

A template can be instantiated in other templates and in packages. When a template `T` is instantiated in another template `U`, it is not mandatory to concretize the required types of `T`. Any required types in `T` that are not concretized upon instantiation in `U` will be propagated to `U`, and will thus become required types of `U`.

On the other hand, when a template `T` is instantiated in a package `P`, every (remaining) required type must be given a concretization. The concrete types may be classes or interfaces from instantiated templates (including the template containing the required type), or from other ordinary Java packages.

Sometimes, it can be nice to provide sensible default concretizations for required types, to alleviate the burden of always having to concretize every (remaining) required type when a template is instantiated in a package. For a mechanism like PT*r*, where a number of required types can be gathered in one template, a way to specify such defaults would indeed be helpful. A default concrete type or implementation could explicitly be given in the declaration of a required type. Another option for simple cases is to choose a default concretization from the bound of the required type. We are still studying how this can best be done, but for simple cases, the prototype compiler currently resorts to the latter approach.

Subtype hierarchies. Table 13.3 shows the relationships that are supported between required types, required interfaces, required classes, classes, and interfaces, for the `extends` and `implements` relations, respectively.

An extends or implements relationship between two required types does not in itself form a hierarchy. Rather, it puts forth a requirement for a hierarchy, i.e. a constraint that actual supplied types must (transitively/reflexively) fulfill.

Constructor definitions. Although it was not explicitly treated in [10], a seemingly common issue with type parameters is that you might want to create objects of the actual types. In Java and Scala, however, this is disallowed, so even if you have a method or a class parameterized by a type `T`, you cannot say "`new T()`".[4] C# is a bit more expressive, and allows the developer to constrain the type parameter by adding a "`where T: new()`" constraint, requiring the actual type to have a parameterless constructor. Other kinds of constructor requirements cannot be expressed.

When utilizing `required` classes, constructor requirements can quite naturally be handled simply by adding the necessary required constructor signatures to the class. These requirements can subsequently be structurally matched with the actual constructors of the class supplied upon instantiation. An example is shown below:

```
template T { required class E { E(int value); } ... }
```

[4] You can, however, in some cases create an object of a generic type `T` using reflection. Furthermore, in Scala, you can utilize implicit factories for similar results.

`extends`	RT	RI	RC	I	C
RT			✓		
RI	✓	✓		✓	
RC			✓		
I	✓	✓		✓	
C			✓		✓
`implements`	RT	RI	RC	I	C
RT					
RI	✓		✓		✓
RC					
I	✓		✓		✓
C					

Table 13.3: *Support for the* `extends` *and* `implements` *relations between required types (RT), required interfaces (RI), required classes (RC), ordinary (template) interfaces (I) and ordinary (template) classes (C). The table is supposed to be read from the top row and down and to the left. I.e.,* `RC extends RT` *is a legal relationship, while the converse* `RT extends RC` *is not.*

Inside classes in the template `T` above, or in classes from other templates or packages that instantiate `T`, statements such as "`new E(42)`" can safely be used.

Note that required constructors can *only* be defined for required *classes*, and not for required interfaces or plain required types.

For a required class `RC` that has a nominal subtyping requirement with bound `B`, where `B` is a class with accessible constructors, the required class must still structurally specify constructor requirements explicitly if "`new RC(...)`" is to be allowed. This is because a Java subclass in general needs not implement the same constructors as its superclass.

Refinement through additions. A required type in PT*r* can be given additions in the same way as classes and interfaces can in basic PT, through an `adds` clause. This can be used to refine constraints in subsequent instantiations. Consider a template `T` defined as follows:

```
template T { required type R { void run(); } }
```

In another template `U` that instantiates `T`, `R` can be refined by adding nominal or structural constraints. An example that does both is shown below:

```
template U {
  inst T;
  required type R adds implements Runnable {void stop();}
}
```

Here, `R` is *refined* in `U`, and further constrained to both nominally implement the `Runnable` interface and to implement a parameterless `stop()` method that returns `void`.

Merging. With basic PT, classes (or interfaces) from different template instantiations can be merged to form one new class. The details of the merge mechanism are beyond the scope of this article, the interested reader is referred to [3] for a more thorough exposition.

In PT*r*, required types can be merged in the same way that ordinary template classes and interfaces can. As for ordinary merges, different *kinds* of types cannot be "cross merged" with each other. I.e. a required interface can only be merged with other required interfaces, required classes only with other required classes, and required types only with other required types. The main difference from ordinary class or interface merging lies in the handling of conflicts. If, in the merge of two required types, there are equal signatures stemming from each of the types, this is not considered a conflict. Rather, the two signatures are merged into one in the resulting required type. If a given pair of equal signatures should indeed be kept separate, the developer may explicitly rename one or both of them in the instantiation. Merging required types where more than one has a nominal bound that is a class is considered a compile time error.

Through merging of required types from different instantiations, the developer is able to express equality constraints across template instances, by explicitly declaring that two previously distinct required types are to be considered the same in the context of the current package or template. In contrast to an ordinary equality constraint, a merge also alleviates the need to provide the same parameter twice, making for more succinct code.

13.4 Fulfilling the Generic Programming Criteria

In this section, we discuss how and to what extent PT*r* fulfills the requirements listed in Table 13.2, as shown in Table 13.4. For brevity, we will not discuss scores that PT*r* "inherits" directly from Java.

Multi-type concepts. The essence of supporting multi-type concepts lies in the ability to simultaneously constrain more than one type. In PT*r*, constraints can be specified by required types within templates, to which several other (required) types of a multi-type concept can refer, and thus be simultaneously constrained. An example of this from our implementation of the generic graph library is shown below:

```
template GraphConcepts {
 required type Vertex { }
 required type Edge { Vertex source(); Vertex target();}
 required type EdgeIter extends Iterator<Edge> { }
 required type OutEdgeIter extends Iterator<Edge> { }
```

	C++	Java	Scala	PT*r*
Multi-type concepts	*	○	●	●
Multiple constraints	*	●	●	●
Associated type access	●	◐	●	●
Constr. on assoc. types	*	◐	●	●
Retroactive modeling	*	○	●	●
Type aliases	●	○	●	○
Separate compilation	○	●	●	◐
Implicit arg. deduction	●	●	●	●
Retroact. concept adapt.	○	○	◐	●
Retroact. constr. adapt.	○	○	◐	●

Table 13.4: *Support for adaptive generic programming in C++, Java, Scala, and PTr.*

```
 required type VertexIter extends Iterator<Vertex> { }
 required interface IncidenceGraph {
   OutEdgeIter out_edges(Vertex v);
   int out_degree(Vertex v); }
 ...
}
```

As we can see from the code, the `Edge`, `VertexIter`, and `IncidenceGraph` types are all constrained by the (same) `Vertex` type, and the `EdgeIter` and `OutEdgeIter` types are constrained by the `Edge` type. Furthermore, we here see an example of traditional Java type parameterization (of the `java.util.Iterator<T>` interface) combined with PT*r*'s required types. Note that this template can be instantiated by relying on default concretization, as discussed briefly in Section 13.3, so that we might e.g. only explicitly concretize `Vertex` and `Edge`, and let the compiler concretize the remaining required types to their bounds. Note also that a concept can be composed from other concepts, each of which might span one or more types, by instantiating templates representing other concepts. Thus, with PT*r* one can express sub-concepts that are themselves comprised of other sub-concepts and/or types, and reuse and/or refine the constraints from these.

Multi-type concepts in Scala are typically implemented through use of the *Concept pattern* [21], parameterized with multiple type parameters. Applying this pattern thus implies creating separate concept classes (or singleton objects) that implement/model the concept interface/trait. Using Scala's `implicit` definitions this can in many cases make for a quite elegant solution. However, for the graph library functionality we found that having multiple multi-type concepts, implemented through the Concept pattern and constrained by the same associated/abstract types, quickly led to a rather complex solution.

Also in Java, multi-type concepts can be expressed through the Concept pattern, but this approach may quickly become cumbersome due to the fact that concept im-

plementations must be referred to explicitly. The score for Java from [10] in Table 13.4 is instead based on a nominal subtyping approach.

Associated type access. Access to associated types is a property that allows code to refer to types that are associated with a generic concept. For instance, the general `Graph` concept has associated types `Edge` and `Vertex`. If the concept is expressed in a package template, and associated generic types as required types, one can simply refer directly (without any additional qualification) to the required types `Edge` and `Vertex`. These required types will be replaced by the actual types upon instantiation, at the latest in a package. Thus, associated type access comes "for free" with PT*r*.

In languages like Java, associated types are typically represented by generic parameters, and this quickly leads to verbose definitions. As an example, contrast the Java definition skeleton in Figure 13.2 on page 271 of the breadth first search algorithm from [10] with the Scala and PT*r* versions directly below it.

Note how in the PT*r* version, the only parameterization that is necessary for the algorithm is to specify the `ColorMap` type, which is not in itself an associated type of the graph concept. The associated (required) type `Vertex` can be accessed directly (even though its actual type has not been supplied yet).

In Scala, an associated type is typically implemented as an abstract type within a class or a trait. Access to such a type is achieved through the type projection construct, e.g. `Graph#Vertex`. For the Scala code above, a parameter for the graph type is thus needed to access the `Vertex` type. Also note that neither the Java nor the Scala versions are parameterized on the `Visitor` concept, as this is not an associated type of the graph concept. For PT*r*, on the other hand, the `Visitor` concept is realized as a required type, and parameterization is thus available without any additional overhead.

A limitation that made implementing graph library functionality a little harder and less natural in Scala was the fact that you cannot use an abstract type as the type of a parameter to a method in another abstract type [20, sec. 3.2.7].

The example presented in Figure 13.2 is closely related to the issue of *constraint propagation*. We notice in that example that the PT*r* version does not need to mention the constraints for generic types it does not itself directly utilize, whereas the Java version must repeat the constrains for `VertexIterator`, `EdgeIterator`, etc. The example below shows how this can lead to complexity in even very simple cases:

Java version:

```
interface VertexListAndIncidenceAndEdgeListGraph<
    Vertex,
    Edge extends GraphEdge<Vertex>,
    VertexIterator extends java.util.Iterator<Vertex>,
    OutEdgeIterator extends java.util.Iterator<Edge>,
    EdgeIterator extends java.util.Iterator<Edge>>
  extends
```

```
VertexListAndIncidenceGraph<Vertex,Edge,
   VertexIterator,OutEdgeIterator>,
EdgeListGraph<Vertex,Edge,EdgeIterator> {}
```

PT*r* version:

```
required interface VertexListAndIncidenceAndEdgeListGraph
 extends VertexListAndIncidenceGraph,EdgeListGraph {}
```

The interface above, in either version, defines nothing more than a composition (through inheritance) of existing interfaces, and does as such not introduce any associated types or requirements on its own. The PT*r* version can be written in a much more succinct manner because there is no need to repeat the associated types as they are *propagated automatically upon instantiation*, and can thus be accessed without resorting to additional generic type parameters.

For Scala, the definition of `VertexListAndIncidenceAndEdgeListGraph` would be similar to the PT*r* version, but its constituents would in each of their definitions have to repeat the constraints for vertices and edges (and the concept could not itself be an associated type, due to the limitation mentioned above).

Constraints on associated types. There are several kinds of constraints that can be useful for associated types. A common form of constraints is that of an equality constraint, i.e. the requirement that an associated type of two other types must be the same. In a template with required types, this can easily be achieved in PT*r* by referring to the same requirement in both of the types. For associated types that were previously unrelated, one can express that they should in a given context be the same by merging the corresponding required types; the new required type will represent the union of the original ones.

Another typical form of constraints are in the form of sub/super relationships. With PT*r*, this can be expressed directly, with e.g. declarations of the form "`required interface I extends J {...}`". If `J` is itself a required interface, the actual type supplied for `I` is constrained to be a subtype of the actual type supplied for `J`.

Basic PT (and thus also PT*r*) amends the problem inherent to Java (and Scala) generics where it is not possible, due to type erasure, to constrain a generic type to two different parameterizations of the same generic interface (or trait). PT allows multiple instantiations (and thus also parameterizations) of a single template.

Retroactive modeling. In C++, retroactive modeling is implicitly supported since the compiler does not check the constraints put forth by templated concepts. Hence, any type can be said to model a concept without any prior reference to the concept itself, as long as the type provides the (implicitly) required operations.

In Java, there is no direct support for retroactively saying that a given class models (implements) a given concept (interface), short of changing its source code. However, a work-around might be the Concept pattern, though this is somewhat awkward to use since concepts must explicitly be passed around.

Java version:

```
class breadth_first_search {
 public static <Vertex,
  Edge extends GraphEdge<Vertex>,
  VertexIterator extends Iterator<Vertex>,
  OutEdgeIterator extends Iterator<Edge>,
  ColorMap extends ReadWritePropertyMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph<
         Vertex,Edge,VertexIterator, OutEdgeIterator> g,
      Vertex s, Visitor vis, ColorMap color) {
    ...
    graph_search.go(g,s,vis,color, ...);
} }
```

Scala version:

```
object breadth_first_search {
 def go[Graph <: VertexListAndIncidenceGraph,
    ColorMap <: ReadWritePropertyMap
          {type Key = Graph#Vertex; type Value = Int}]
       (g: Graph, s:Graph#Vertex, vis: Visitor,
          color: ColorMap ){
    ...
    graph_search.go(g, s, vis, color, ...);
} }
```

PT*r* version:

```
inst GraphConceps;
class breadth_first_search {
 public static <ColorMap extends
     ReadWritePropertyMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph g, Vertex s,
      Visitor vis, ColorMap color) {
    ...
    graph_search.go(g,s,vis,color, ...);
} }
```

Figure 13.2: *Associated type access in Java, Scala and PTr*

As Java, Scala supports retroactive modeling through the use of the Concept pattern, but `implicit` declarations make the use of concepts much more convenient and natural to the programmer in Scala.

Existing Scala libraries can be extended (or rather, appear to be extended) through the "library pimping" approach [19], in order to support retroactive modeling. However, this approach is typically based on implicit runtime creation of new objects, which might lead to subtle bugs e.g. when references to such objects are passed around. There is, in our opinion, a significant difference between retroactively adjusting the model (as PT*r* can do), and annotating the model with conversions to and from the modeled concepts.

With PT*r*, classes from templates can model new concepts through being merged with other classes that model the concept, or by having interface implementation declarations added by an `adds` part. The possibility for name changes makes it easier to let existing code retroactively model new interfaces. A small example sketch is shown below, where a concept `M` is realized by the interface `M`. The template `T` contains a class `C`, that implements the desired functionality in a method `mx`, however, it does not explicitly implement `M`:

```
interface M { void m(); }
template T { class C { void mx() { ... } } }
```

With PT*r*, we can retroactively define the implements-relation between `M` and an instance of `C`, as follows:

```
// rename method "mx" to "m":
inst T with C => C ( mx() -> m );
// add the interface implementation decl:
class C adds implements M { }
```

Type aliases. Type aliases are supported by C++ and Scala (and other languages) as a way to make long type names shorter, and are as such especially useful when dealing with heavily parameterized code. However, it has not been our goal to support this in our work with PT*r*, and the level of support is thus the same as for plain Java. Even so, PT*r* does alleviate this issue to a certain extent, since the parameterized types can be fixed at the time of instantiation, and need thus not be repeated for subsequent uses.

Separate compilation. This criterion includes both separate type checking and compilation into independent units. Java and Scala support both parts of the criterion, while the templates of C++ supports neither. The former part is fulfilled by PT*r*, since every template can be separately type checked independently of subsequent usage. The latter part is not supported by the current prototype compiler, which produces separate code for each instantiation (a heterogenous implementation). However, we have previously experimented with how a homogenous implementation can be made for an extended JVM, with special instructions e.g. for invoking methods in adapted template

classes. Such an approach does, in contrast to the current heterogenous compile-time specialization scheme, come with some runtime performance overhead (which is also incidentally true for Scala's implicit definitions and Java's runtime casts due to erasure).

Retroactive concept adaption. As one of the additional two criteria we added in order to support the programming of adaptable generic libraries, retroactive concept adaption is the ability to unintrusively (i.e. without making changes to the original source code) make certain changes to a concept after its initial definition. These changes include renaming of methods and of the concept itself, and changes to the types returned by or accepted as parameters to the operations (methods) of the concept. Such changes can be important in order to provide a better match for existing code, or in order to better reflect the domain of the problems that the program is supposed to solve. Name changes may seem like a trivial modification to any program, but influential development methodologies like domain-driven design (see e.g. [8]), as well as research into naming conventions and intended semantics (see e.g. [12]), put emphasis on the importance of proper naming. Neither Java nor C++ supports the retroactive adaption of concepts, and developers are hence relegated to either make do with the concept as they were originally defined, make wrappers around them, or duplicate code. In Scala, concepts can, to a certain extent, be retroactively adapted through the use of implicit declarations and inheritance.

For PT*r*, retroactive adaption is one of the main motivating goals for the mechanism, and adaption of concepts (defined as interfaces, required interfaces or abstract base classes) as well as their potential implementations can be expressed as part of an instantiation of a template. The fact that each instantiation results in a new set of classes is the main reason for PT*r*'s flexibility with regard to name changes.

Beyond name changes, a concept in PT*r* might be refined by adding new operation signatures through the use of `adds` clauses (for both concept definitions and implementations). The PT*r* approach supports overloads and (single) dynamic dispatch of added operations, in contrast to mechanisms such as extension methods in C# or the Concept pattern through implicits in Scala, which rely on static dispatch.

Retroactive constraint adaption. Constraints form a significant part of the interface to a generic library, and hence if retroactive modeling and adaption are deemed important, the possibility for retroactive adaption of constraints should be of equal importance.

In Java, constraints cannot be adapted short of changing the source code or creating wrappers that refine the original constraint. Scala supports refinement of constraints expressed as abstract types through subtyping, but the original constraint cannot be adapted.

With PT*r*, constraints in form of required types can be adapted in several ways. To begin with, their names can be changed, along with the names of method signatures within them. Changing the name of a required type might be useful when a type is not supplied at instantiation, and a default interface definition is subsequently created by

the compiler. Changing the names of method signatures is useful in order to adapt the generic library to existing code, when one is unable or unwilling to change the latter. Below is an example of a small library for representing cities and roads, encapsulated in a package named `Geography`.

```
package Geography {
  class City { String name; int population; ... }
  class Road {
    private City from, to;
    City getFrom() { return from; }
    void setFrom(City c) { from = c; }
    City getTo() { return to; }
    void setTo(City c) { to = c; }
} }
```

It would be nice to be able to apply the algorithms from the generic graph library, like e.g. Dijkstra's shortest paths, to cities and roads from the `Geography` package. In our implementation, the `Algorithms` template contains the desired functionality, and it will in turn instantiate the `GraphConcepts` template, which contains the requirement for a `Vertex` class and an `Edge` class, and corresponding constraints. The requirements will be propagated to the instantiating package. Thus, we can use the instantiation below to adapt the generic constraints to our `Geography` package:

```
inst Algorithms with Vertex <= City,
   Edge (source() -> getFrom, target() -> getTo) <= Road;
```

With this approach, we can apply all the algorithms from the generic graph library to the classes from the `Geography` package.

A constraint may also be adapted in PT*r* by providing an addition that refines the constraint, typically in a narrowing fashion. For instance, to further constrain the `Vertex` type to include a `getName` method, we can utilize the following code in a template:

```
inst GraphConcepts;
required type Vertex adds { String getName(); }
```

It is important to note that such modifications are local to the current instantiation, and do not propagate globally to other potential instantiations of the `GraphConcepts` template in other parts of the program. Thus, retroactive constraint adaption can in PT*r* be done in a controlled and unintrusive manner.

Aside: code complexity comparison. The Java implementation from [10] has 760 lines of code[5], while the corresponding PT*r* implementation has 691 lines. However, this includes a lot of imperative code that is identical in the two versions. To approach

[5]Counted with the CLOC tool: `http://cloc.sourceforge.net/`

an understanding of the relative complexity in terms of parameterizations and constraints, we have tried to count these in a reasonable way. In both versions we have counted all elements that occur within angle brackets <...>, and all elements that occur as subtype bounds for generic constraints. In addition, we have counted all required types and explicit concretizations of such for PT*r*. The count is 542 occurrences for Java and 325 for PT*r*, i.e. a reduction of about 40%. We think that this can have a significant impact on the comprehensibility and maintainability of the code.

13.5 Related Work

Virtual classes originated with the BETA language [16, 17], and has subsequently inspired a host of other languages and mechanisms, such as e.g. gbeta [6], Caesar [1], J& [18], and Newspeak [4]. These mechanisms support a certain degree of parameterization based on overrides (or *refinements*) of the virtual types, typically contained within ordinary classes. An important advantage of the virtual type approach over type parameterization as found e.g. in Java is the automatic propagation of constraints.

In [25], Thorup argues for the inclusion of virtual types in Java as an alternative to classes with type parameters. In this proposal, Java classes can contain bounded `typedef`s, that can be used to define generic classes that abstract over an open set of types. A problem with this approach is that static type safety is reduced, and every class could now potentially be subject to runtime exceptions due to covariant subtyping. With PT*r*, this kind of typing issues are not problematic since actual types *substitute* all occurrences of references to required types at *compile-time*.

A subsequent paper [26] presents an approach that combines virtual types (in a type safe variant [27]) with structural subtyping. This gives three "dimensions" of subtyping: the ordinary subclass variant, covariance in the generic parameter, and binding of the generic parameter to its bound. This allows virtual types to be used in many situations where parameterized types traditionally have been considered a better option. We have not discussed the issue of subtyping of parameterized classes to any extent in this paper, but it seems clear that if subtype relations between different parameterizations of the same class are required, PT*r* is not the ideal tool, since every instantiation results in a new, independent, set of classes (though template classes can implement common external interfaces).

Both [25] and [26] provide adequate support for associated types of concepts, however, as opposed to in PT*r*, multi-type concepts are not as easily expressed.

In [13], the authors introduce explicit support for associated types and constraint propagation in C# with a mechanism resembling virtual classes. However, an important distinction from prototypical virtual classes is that nested types are not specific for each object of an outer class. Like PT*r*, they support assignment of pre-existing types to the associated (virtual) type definitions. Constraint propagation is automatic, as for PT*r*, but limited to the confines of singular class hierarchies.

Neither of the virtual type-based mechanisms support retroactive adaption of concepts or constraints in a manner resembling PT*r*.

Scala [20] has been discussed in some detail in this paper, however, there are some additional points that should be addressed. We have mentioned that the static nature of our mechanism facilitates a simpler type system, and arguably also a simpler conceptual model. However, this has some obvious drawbacks, most notably that full family polymorphism [7] and dependent types are not supported (since we do not allow creating instances of templates at runtime). Furthermore, Scala supports several advanced features that PT*r* does not, such as e.g. higher-ordered types, pattern matching and implicits.

JavaGI [28] is an extension to Java inspired by Haskell's type classes. It supports retroactive modeling through explicit implementation declarations. *Multi-headed interfaces* provide support for multi-type concepts in a natural way, exemplified by the Observer pattern [9]; a corresponding implementation in PT could be a template with types for each role, an example can be found in [2]. JavaGI does not fully support retroactive adaption.

The $\mathcal{G}$ language [24] compiles to C++ and contains explicit support for generic concepts and models, and supports all the criteria listed in Table 13.2A. Like PT*r* it supports modular type checking, and also separate compilation, but not the criteria from Table 13.2B.

Partly building on the work on $\mathcal{G}$, ConceptC++ [11] supports explicit concept definitions and constrained function and class templates. Retroactive modeling is achieved through concept maps, and in general the criteria from Table 13.2A are well supported, though type checking is not fully modular due to their support for concept-based overloading, which PT*r* does not fully support. A goal for ConceptC++ was to form the basis for the inclusion of concepts in the new C++0x standard. However, the current version of the standard excludes concept support.[6] To our knowledge, neither ConceptC++ nor C++0x fully supports the criteria in Table 13.2B.

13.6 Concluding Remarks

The package template (PT) mechanism extended with required type specifications yields a, to the best of the authors' knowledge, rather novel blend of support for parameterization and retroactive modeling and adaption through compile-time specialization with separate type checking. We refer to this variant of PT as PT*r*.

We have shown that PT*r* applied to a mainstream OO language like Java supports almost all of the criteria put forth by [10], as well as additional criteria identified in this paper for adaptability of generic code. Furthermore, early investigations suggest that PT*r* can provide a significant reduction of duplicated code in generic Java libraries, and in some cases provide a simpler solution compared to other mechanisms aiming at similar problems, such as virtual class- or abstract type-based mechanisms.

[6] See e.g. `http://drdobbs.com/architecture-and-design/218600111?pgno=3`

Acknowledgements

This work has been done within the context of the SWAT project (NFR grant 167172/V30). We thank the reviewers for helpful comments on previous versions of this paper, and Steinar Kaldager and Daniel Rødskog for dedicated work on the PT(*r*) compiler.

Bibliography

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 37–42, New York, NY, USA, 2009. ACM.

[3] Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. Challenges in the design of the package template mechanism. *Transactions on Aspect-Oriented Programming*, 2012. To appear, available now from `http://swat.project.ifi.uio.no/`.

[4] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In Theo D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 405–428. Springer, 2010.

[5] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good match for object-oriented programming languages. In *ECOOP '97*, 1997.

[6] Erik Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[7] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 Budapest, Hungary, June 18-22, 2001, Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[8] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17:145–205, March 2007.

[11] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proc. OOPSLA '06*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[12] Einar Høst and Bjarte Østvold. Debugging method names. In Sophia Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317. Springer Berlin / Heidelberg, 2009.

[13] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 1–19, New York, NY, USA, 2005. ACM.

[14] M. Jazayeri, R. Loos, and D. Musser. Generic Programming - Report from Dagstuhl Seminar. Technical report, 1998.

[15] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[16] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*, pages 397–406, New York, NY, USA, 1989. ACM.

[17] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, New York, NY, USA, 1993.

[18] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, NY, USA, 2006. ACM.

[19] Martin Odersky. Pimp my library, 2006.

[20] Martin Odersky. The scala language spec. version 2.9 – draft, 2011.

[21] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proc. OOPSLA 2010*, pages 341–360, New York, NY, USA, 2010. ACM.

[22] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[23] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, December 2001.

[24] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423 – 465, 2008. Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).

[25] Kresten Krab Thorup. Genericity in java with virtual types. In *In Proceedings ECOOP '97*, pages 444–471. Springer-Verlag, 1997.

[26] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 186–204, London, UK, 1999. Springer-Verlag.

[27] Mads Torgersen. Virtual types are statically safe. In *In Proceedings of FOOL '98: 5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[28] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized Interfaces for Java. In *Proc. European Conference on Object-Oriented Programming 2007*, LNCS. Springer-Verlag, July 2007.

Chapter 14

Paper VII: Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code

Authors. Eyvind W. Axelsen and Stein Krogdahl

Publication. Proceedings of the 11th International Conference on Generative Programming and Component Engineering, 2012 (GPCE'12) .

Abstract. Package Templates (PT) is a mechanism designed for writing reusable modules, called templates, each consisting of a set of classes that can be adapted to their use in a program through compile-time specialization. A template must be *instantiated* in a program before its classes can be used. The mechanism supports type-safe renaming, merging, type parameterization and refinement in the form of static additions and overrides that are orthogonal to the corresponding concepts of ordinary inheritance.

In this paper, we consider PT as an extension to Java, and a PT program will then consist of a number of Java packages and templates, where templates are instantiated in packages or other templates. Our aim and main contribution is to define the meaning of such a program, and to show that this definition is consistent. We first show this for a core subset of PT, C-PT, and define a set of source-to-source transformations for converting C-PT programs to plain Java programs using semantics we have described informally in previous papers. We can then define the meaning of a C-PT program in terms of the resulting Java program. Thus, we have to verify that the transformations will always convert a legal C-PT program to a legal Java program. Finally, we briefly discuss how this approach can be extended to full PT.

A main challenge is to preserve externally visible names (for classes, methods and fields), and at the same time prevent unwanted subsequent rebindings caused e.g. by overload resolution in the Java compiler. Names that are bound to declarations in a

template should not be rebound to different declarations by subsequent compositions or adaptions.

In addition to defining the runtime semantics of PT constructs in terms of their translation to Java, the transformation rules can also be seen as a high-level approach to how a compiler for this language might be implemented.

Categories and Subject Descriptors. D.3.3 **[Programming Languages]**: Language Constructs and Features — *Classes and Objects* D.3.4 **[Programming Languages]**: Processors — *Code generation*

General Terms. Languages

Keywords. Templates, Transformations, Java

14.1 Introduction

Package Templates (PT) [5, 8, 19] is a mechanism intended for writing reusable and adaptable modules (called templates), where each module consists of a collection of classes. PT is based on compile-time instantiation and specialization, and allows type-safe renaming, merging, parameterization and refinement in the form of static additions and overrides that are orthogonal to the corresponding concepts of ordinary inheritance.

While PT is intended to be viable for a broad range of object-oriented languages, we will in this paper consider PT as an extension to Java. A PT program will then consist of a number of Java packages and a number of templates that are specified for inclusion in the packages. The PT mechanism, as described in previous papers, includes a number of conditions that must be met for such a set to be a legal PT program.

An important aim of this work is to present a more formal definition of the semantics of the PT language than those of previous papers. We therefore define a series of *source-to-source* transformations that weave together the templates and packages of a PT program. Our claim is that the result of these transformations will be a legal Java program, and we can thus define the meaning of a legal PT program as that of the corresponding Java program. To substantiate this claim, we present a lemma for each transformation, and a corresponding proof sketch for the correctness of each lemma. We also argue that if all the lemmas hold, the result from the transformations will indeed be a legal Java program.

Since our previous works have not established a formal definition of PT, we cannot prove a one-to-one correspondence between our transformation definitions and previous intuitive descriptions. However, we aim to faithfully represent these previous descriptions in the transformations in this work.

A main design goal of PT is to preserve the semantics of individual templates as they are composed with other templates. For instance, unintentional overrides or changes in method selection by the standard overload resolution mechanism of Java as a result of composition should not happen. We thus aim for *behavior preservation* [24] (disregarding reflection) of the original templates in the transformations we will describe, i.e. that the composed program contains *"semantically equivalent references and operations"* [24, p. 40], unless explicitly overridden by the programmer. This can also be expressed as preserving the semantic bindings of the original template, and in the rest of this paper we will use the term *binding preservation* for this property. Preserving bindings through composition is different from what is done in typical feature-oriented approaches such as e.g. ATS [9] or FeatureHouse [2], where rebinding post composition is the norm.

In addition to binding preservation we also want *name preservation*, i.e. that externally visible names (names that might be referred to in client code) are preserved through the transformations unless explicitly renamed by the programmer of the client code.

The translation of PT code to Java code thus raises many of the same problems as those of refactoring object oriented code (see e.g. [14, 20, 21, 23, 24, 26]), but since

package templates allow a wider array of changes and specializations than those traditionally offered by refactoring (e.g. class merging and refinement), there are quite a few interesting and somewhat novel challenges involved.

Another aim for this work is to transform PT code to *efficient* Java code, so that the transformations could be used as a basis for an actual implementation of PT. Thus, we want to create plain Java classes, without resorting e.g. to wrappers or runtime dictionaries, in contrast to existing implementations of many mechanisms with some resemblance to PT, such as e.g. Expanders [28] or JavaGI [29].

To keep the exposition relatively short, we will focus on a core subset of PT in this paper. Section 14.5 gives a short, intuitive, description of what would be needed in order to handle full PT.

Contributions The contributions of this paper can be summarized as follows: 1) we define the semantics of the core of the PT language as applied to Java, including a set of lemmas with a proof sketch showing that these transformations will always result in a legal Java program; 2) we present an approach to source-to-source transformation of compositional Java code that preserves bindings and visible names; 3) we present an implementation strategy for PT.

Relation to previous PT papers The main ideas of PT were first described informally in [19], and several of the important design decisions were discussed in more detail in [8]. In [5] we describe an extension of PT for generic programming. These papers focus on design issues and extensions of PT, along with example code motivating the design choices. This paper, on the other hand, does not introduce any new concepts to PT, but instead aims to clearly define the meaning of PT programs in terms of their translation to plain Java.

A prototype compiler based on JastAdd [13] supporting most of the concepts of PT is available from `http://swat.project.ifi.uio.no`. An implementation for and in the dynamic language Groovy was discussed in [4, 6], while an implementation for and in Boo is informally described in [27].

14.2 Brief Overview of Core PT

In this section we present a general, intuitive, overview of a subset of the basic PT mechanism that we will refer to as *Core PT* (C-PT). This subset will be used for most of this paper. It is designed to simplify the following exposition, while at the same time retaining most of the distinguishing characteristics of full PT. Later sections will provide more detail on the exact semantics of the constructs demonstrated in this section.

A package template in C-PT looks much like a regular Java package, but we will use a syntax where curly braces enclose the contents of both templates and regular packages, e.g.:

```
template T {
  class A { ... }
  class B extends A { ... }
}
```

The contents of a template in C-PT, i.e. everything between "`template` <*NAME*> `{`" and the corresponding final "`}`", is a collection of valid Java classes, with the following exceptions:

- Nested class definitions are not allowed. Exempt from this rule are anonymous nested classes created within method definitions.
- Access modifiers (`public, private, protected`) are not used. Every definition thus has default access.
- Explicit constructor declarations in template classes are not allowed (but calling the implicit constructors, e.g. `new C()`, is permitted).
- Static declarations are not allowed, neither are references to static declarations (methods, variables), e.g. from the Java standard library.
- Generic class or method definitions are not allowed.
- Covariant return types in the signatures of method overrides are not allowed.
- In addition to ordinary classes, templates may also contain:
  - Instantiations of other templates in the form of `inst` statements (see below).
  - Additions to classes from instantiated templates declared with an `adds` clause (see example below) in the instantiating template/package; this construct is called an *addition class*. For each instantiated template class, there can be at most one addition class, and the addition class must have the same name as the instantiated template class.

Note that full PT allows interfaces and enumerators as well as classes in templates, but this is not included in C-PT.

In contrast to for instance templates in C++, package templates can be type checked independently of their potential usage(s).

A template is *instantiated* at compile-time with an `inst` statement, see Figure 14.1 for syntax. Instantiations create a local copy of the template classes, potentially with specified modifications, within the instantiating package or template. The instantiations preserve any class hierarchies found in the template, and modifications can be made at any level in such hierarchies. The order of instantiations, classes and addition classes is not significant. Cyclic instantiations are not allowed, and templates in C-PT cannot refer to other packages that have `inst` statements. An example usage is shown below:

inst-stmt ::= **inst** *template-name* [**with** *rename-list*] ;
rename-list ::= *rename-clause* (**,** *rename-clause*)*
rename-clause ::= *class-name* **=>** *class-name* [**(** *attrib-rename* (**,** *attrib-rename*)* **)**]
attrib-rename ::= *attrib-name* [**(** [*type-name* (**,** *type-name*)*] **)**] **->** *attrib-name*

Figure 14.1: Syntax for the `inst` statement. Non-terminal symbols and meta symbols (parentheses, brackets and asterisks) are written in *italics*, while terminal symbols are written in **bold**. Note that parentheses occur as both meta symbols for grouping and as terminals.

```
package P {
  inst T with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D ext. C since B ext. A
}
```

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `P`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. "`inst T`". However, modifications can also be made to the template classes upon instantiation, such as:

- Elements of the template may be renamed. This is specified in the `with`-clause of the `inst`-statement, and is only shown for class names above (`A` is renamed to `C` and `B` is renamed to `D`). For renaming of class attributes another arrow is used (`->`). In Section 14.4 we get back to the language rules that ensure that renames do not cause problems with regard to name clashes, accidental rebindings or overrides, etc.
- In each instantiation the classes in the template may be given additions: fields and methods may be added and virtual methods may be overridden. This is done in `adds`-clauses as shown for `C` and `D`. The classes to which `C` and `D` adds are referred to as their tsuper-classes. (Note, however, that the term "tsuper-class" is utilized only as a technical term when describing the transformations, and that such classes do not exist at runtime, in contrast to ordinary super-classes.) As for renames, we shall consider the actual rules and semantics for additions in Section 14.4.

Classes from different template instantiations may be *merged* to form one new class. Syntactically, merging is obtained by renaming classes from two or more template instantiations to the same name, and they thereby end up as one class. The new class gets all the attributes of the instantiated classes, together with the attributes of the common addition class. Consider the simple example below:

```
template T { class A { int i; A m1(A a) { ... } } }
template U { class B { int j; B m2(B b) { ... } } }
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;

class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the method `m2` from `B` is overridden in the `adds` clause, and furthermore how both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`.

14.3 Overview of the Approach

Our overall aim is to provide a consistent definition of the semantics of a C-PT program through transformation on the source code, ultimately ending up with plain Java source that defines the dynamic semantics of the program.

We start by describing some classifications for different kinds of templates and packages.

14.3.1 Closed and Open Templates and Packages

Closed templates In the following, we shall refer to templates that do not instantiate any other templates as *closed templates*. Thus, closed templates are semantically complete and self-contained (with the exception of potential references to external Java packages) units that can be separately type checked. Moreover, a closed template in C-PT is a true subset of plain Java, as long as we remove the opening "`template <NAME> {`" declaration along with the final closing "}". This allows us to utilize any open compiler framework for Java in an implementation of the following transformations, such as e.g. JastAdd [13] or Polyglot [22]. In the transformations defined in the Section 14.4, we will just assume that a correct plain Java AST with semantic bindings of any closed template is available for the transformations to utilize, and when we refer to the *binding* (or semantic binding) of a variable usage or method call, we mean the binding to the corresponding declaration that can be obtained from the AST.

Open templates An *open template* defines the legal template definitions of PT. An open template is a closed template that may also contain template instantiations (`inst` statements) and addition classes.

An open template is thus *not* self-contained; in order to type check an open template `T` we need access to the templates that are instantiated by `T`.

Open and closed templates are concepts that are meaningful and well-defined both for the full PT language and for C-PT.

Open and closed packages Corresponding to the concepts of open/closed templates, we can also have open and closed packages. A *closed package* is an ordinary Java package. An *open package* is a package that contains template instantiations, and potentially also addition classes as well as ordinary classes.

14.3.2 Instantiations

Arguably, the most important operation in PT is the (compile-time) instantiation of templates. In this paper we will describe this operation as an iterative transformation where the starting point is an open template/package performing the instantiation of a set of closed templates. Our goal is then to show that the result of this transformation will be a correct, closed template or package, and that the externally visible names from the instantiated template(s) are preserved, unless renaming is explicitly specified.

Presuming that we are able to show this, we can then view a PT program as a tree of instantiations, where an instantiation corresponds to an edge and an instantiated template or a package to a node. The root node is thus an open package, and the leaf nodes are (by definition) closed templates. The inner nodes will correspond to instances of open templates. We can then iteratively transform the templates, working from the leaf nodes and in towards the root. For each transformation, a set of leaf nodes will be transformed to become a part of their instantiator, which then becomes a new leaf node in closed template form. When only the package remains, we will know that this package is a correct, closed package with the expected names.

Based on this, we can define the dynamic semantics of a PT program (a set of templates and a package) as that of the resulting plain Java program (a closed Java package).

14.3.3 Problems and Example Programs

In this section we will consider just a few examples of some of the things that might typically break semantic bindings with a naïve approach to source-to-source transformation, even of seemingly innocuous programs. Our transformations must be designed so that they avoid such pitfalls.

Unintentional method call rebinding When adding methods to an existing class in source code form, existing call sites might unintentionally be rebound to a different method after subsequent compilation by a standard Java compiler. As an example, consider the following template `T1` and package `P1`:

```
template T1 {
  class A {
    void m(Object o) { ... }
    void f() { m("Hello World"); }
  }
}
```

```
package P1 {
   inst T1;
   class A adds { void m(String s) { ... } }
}
```

When performing a semantic check of the template `T1` in isolation, the call to `m` in `f` will bind to the only definition of `m` available, taking an `Object` as parameter.

The semantics of the `adds` clause for `A` in `P1` is that its contents should be added to the class `A`. Thus, one could suggest defining the the resulting closed package `P1` as follows:

```
package P1 {
  class A {
    void m(Object o) { ... }
    void f() { m("Hello World"); }
    void m(String s) { ... }
  }
}
```

However, the source code for `A` shown directly above does not preserve the behavior of `A` as it was defined in the template `T1`, since the call to `m` in `f` will now bind to the overload taking a `String` instead of the original version taking an `Object`.

Unintentional variable rebinding As for methods, variables may also be rebound or shadowed when renaming or adding new declarations to a class. Consider e.g. the following template `T2` and package `P2`:

```
template T2 {

  abstract class X {
     int i = -1;
     abstract int f();
  }

  class A {
    int j = 42;
    int m() {
      return new X() { int f() { return j; } }.f();
    }
  }
}

package P2 {
  inst T2 with X => X (i -> j);
}
```

In the instantiation of `T2` in `P2`, the name of the variable `X.i` is changed to `j`. However, this would, in a direct translation to Java, cause the implementation of `f` in the anonymous subclass to return `-1` instead of `42` as originally intended.

Unintentional method overrides Since Java does not have explicit mandatory syntax for defining virtual methods and their overrides, naïvely combining class hierarchies might lead to redefinition of what is an override and what is an original definition. Consider e.g. the following template `T3` and package `P3`:

```
template T3 {
  class A { }
  class B extends A { void m() { ... } }
}

package P3 {
   inst T3;
   class A adds { void m() { ... } }
}
```

Here, the method `m` in `T3.B` is not an override of any other method. However, with the introduction of `m` in `P3.A`, it would become an override of that method.

In the following section we will consider ways to transform templates that take problems like those above into account, and thus preserve the semantics of the composed templates.

14.4 Transformations

In this section we will consider the transformation for converting a C-PT program consisting of an open package and a set of open and closed templates to a closed Java package. If a package or template instantiates another template `T`, `T` must be part of the set of templates in the program.

The full transformation is divided into four phases or sub-transformations that are to be performed sequentially: The *fortifying transformation* makes a template more resilient when involved in further composition and adaption, the *renaming transformation* performs renaming of elements in a template according to the corresponding `inst` statement, the *addition-handling transformation* prepares individual templates for composition with the instantiating template/package, taking addition classes into account, and the *composing transformation* performs the actual composition of the classes from instantiated classes with the addition classes of an open template/package. For each of these transformations we formulate a lemma, and sketch a proof for its correctness.

Renaming program elements When a declaration is renamed as part of a transformation, the corresponding usages of this declaration are also updated accordingly, based on established semantic bindings, even if we do not explicitly state this in the text.

Fresh names A fresh name is a name that is guaranteed to be unique across the entire program, e.g. based on a strictly increasing monotonic integer generator. When defining the transformations, we just assume that a function for generating fresh names is available.

14.4.1 The Fortifying Transformation

This transformation consists of a number of steps that can be applied to any closed template in C-PT, without taking any potential instantiations of said template into account. The input to the transformation is a fully type checked closed template in C-PT. The purpose of the transformation is to make the template code more robust, in the sense that it is resilient to unintentional rebindings when additions or renames are performed by subsequent transformations, resulting in a program text that is ultimately to be handed off to a Java compiler while preserving PT semantics.

1. Every variable declaration within a method definition, including the method's formal parameters, are given fresh names.
2. (a) Every method override or implementation stemming from an original definition in a superclass (abstract or concrete), is marked with an `@Override` annotation, if such an annotation is not already present.
   (b) Then, every method declaration in anonymous inner classes that are not marked with `@Override` is given a fresh name.
3. For every method call to a method `m` with formal parameters $p_1, ..., p_N$ of type $t_1, ..., t_N$, $N \in \mathbb{N}$, the actual parameters $ap_1, ..., ap_N$ in the call are prefixed with an explicit cast to the type of the formal parameter, if such a cast is not already present. I.e., the call to `m(`$ap_1, ..., ap_N$`)` will be transformed to `m((`t_1`)` ap_1 `, ..., (`t_N`)` ap_N`)`.
4. For every usage of an instance variable `v` in the body of a class `C`:
   (a) If `C` is an anonymous inner class of an enclosing class `Y`, and the variable usage is unqualified (i.e. not of the form `r.v`) and binds to a variable declared in `Y` or any of Y's superclasses, refer to the declaring class of `v` as `X`. Then transform the variable usage to `((X)Y.this).v`.
   (b) If the usage is unqualified and binds to a variable declared in `C` or one of its superclasses, refer to the declaring class of `v` as `X`. Then transform the usage of `v` to `((X)this).v`.
   (c) If the usage is qualified (i.e. it is of the form `r.v`) refer to the declaring class of `v` as `X`. Then enclose every existing qualification `r` in parenthesis containing a cast, so that the full usage becomes `((X)r).v`. Note that `r` itself may be a composite reference, and that this transformation thus must be done recursively on every part of this reference.

Note that casts and qualifications added by the transformation steps will be subject to the remaining transformations in the same way as other names in the program. Fur-

thermore, note that all casts added are upcasts, which, if used in an implementation, would be performance-neutral.

Lemma 1. *Applying the fortifying transformation on a closed template in C-PT results in a closed template in C-PT with 1) the same externally visible names (name preservation), and, 2) the same semantics and bindings (binding preservation).*

Proof sketch In order to prove that the Lemma 1 is correct, we need to check that both property 1 (name preservation) and property 2 (binding preservation) is upheld by each transformation step. We therefore consider each of the numbered transformation steps of the fortifying transformation:

1. Property 1 is upheld since local variables and method parameters are not externally visible names. Furthermore, changing the name of a local variable or parameter does not cause any externally visible effects. Property 2 is upheld since the renaming to a fresh name guarantees that no (new) name clashes will occur, and a rename based on semantic bindings guarantees that there will be no accidental rebindings to other names.
2. (a) Property 1 is upheld since annotations do not change visible names. Property 2 is upheld since the `@Override` annotation has no effect on the runtime semantics of a Java program (disregarding potential reflective programs checking for such an annotation).
   (b) Property 1 is upheld since methods in anonymous inner classes that are not overrides are not externally visible. Property 2 is upheld since the new name is a fresh name, and the renaming is based on the established bindings.
3. Property 1 is upheld since this step does not change any names. Property 2 is upheld since the casts added are upcasts to the types of the formal arguments from the method definition, thus it will have no effect (at this stage) to add such casts, other than to make overload resolution explicit.
4. (a) Property 1 is upheld since there are no renames performed by this transformation. Property 2 is upheld since the declaring class is readily available from the variable's binding, as is the property of being an anonymous inner class or not. The qualification of `this.X` is thus safe to use for any inner class to refer to an outer class. An upcast to the declaring class, which might differ from the syntactically enclosing outer class, will not change the binding.
   (b) Property 1 is upheld since there are no renames performed by this transformation. Property 2 is upheld since the declaring class is readily available from the variable's binding. The qualification of `this.` is safe to use to refer to an instance variable of the current class. The cast to the declaring class is safe since the class is statically known and upcasting has no runtime effect.
   (c) Property 1 is upheld since there are no renames performed by this transformation. Property 2 is upheld since the declaring class is readily available

from the variable's binding. The cast to the declaring class is safe since since the class is statically known and upcasting has no runtime effect.

□

Strongly closed templates We will say that a closed template in C-PT is *strongly closed* when all the steps of the fortifying transformation have been performed.

14.4.2 The Renaming Transformation

The `inst` statement may specify renames; its syntactical form is shown in Figure 14.1.

Renames of classes, methods, and fields are performed based on the semantic bindings of strongly closed templates in C-PT. The renaming transformation thus consists of only one step: perform renames of relevant declarations as specified in the `inst` statement, and change all occurrences bound to these declarations accordingly. However, before the renaming transformation can be performed, the preconditions below must be upheld. These preconditions are based on the rules of PT.

Preconditions for renaming First, it is important to note that one can only rename a field or a method in the class in which it is textually defined, not in a subclass.

For renaming of a *field* named `f` defined in a class `C` of a closed template `T` in C-PT to a new name `f'`, the following single rule must be followed by the programmer:

- There cannot be another field named `f'` in `C`, unless this field is also renamed.

A *method* named `m` defined in a class `C` of a closed template `T` can be renamed to a new name `m'` if the following rules are followed:

- The method `m` cannot have an `@Override` annotation. (If it does, this method is not the original definition of the method, and the rename must thus be performed at a higher level in the class hierarchy.)
- It is not allowed for a method rename to cause a method to become an override of another, previously unrelated, method. Thus, the following condition must be fulfilled both for `C` and recursively for any subclasses of `C` in `T`: If there is one or more methods named `m'` in `C` or any of its superclasses, then the signature of `m` must be such that it will represent a legal overload, according to Java rules, with respect to the existing methods named `m'`.

A *class* named `C` defined in a template `T` can be renamed to `C'` if the following rules are adhered to (if a class is not renamed explicitly in the `inst` statement, we consider this an implicit rename to the same name, e.g. `C => C`):

- There cannot be another class named `C'` in `T`, unless this class is also renamed.
- There cannot be another class named `C'` defined directly in the instantiating template/package, unless the other class is an addition class.

Lemma 2. *Performing legal renames according to the rules above on a strongly closed template in C-PT results in a strongly closed template in C-PT with the exact same semantics and bindings, modulo renamed methods, variables, and classes.*

Proof sketch After the renames, the template will still be syntactically identical (except for names) to the strongly closed one we started with. However, at the outset, rebindings could now have occurred, and these could even have made the template semantically illegal. We thus have to show that, if the PT-restrictions for renaming are followed, and because we started with a strongly closed template in C-PT, such rebindings will not occur.

Fields: When renaming a field `f` to a new name `f'`, rebinding could occur if there are references to another field `f'`, now binding to the newly renamed field instead, or the other way around. The renaming preconditions explicitly forbid two fields in the same class to share a name, so the problems will only occur with fields in other classes, i.e. sub- or superclasses. However, since the fortifying transformations make every field reference explicit with casts to the defining class, this will not be a problem.

Methods: The preconditions guarantee that the new method name `m'` will either be a name that is not already used for a method in the class hierarchy of `C` in `T`, or the new method will be an overload of an existing method named `m'`. The fortifying transformations make sure that no rebinding will take place even if a new, potentially better matching, overload is introduced. The preconditions make sure that no new overrides are introduced.

Classes: The preconditions guarantee that there will be no name clashes with existing classes in `T` or in the instantiating package/template (merges will be handled in Section 14.4.4). Since static references and definition are disallowed in C-PT, there can be no confusion between variable names and class names. □

14.4.3 The Addition-Handling Transformation

The purpose of the addition-handling transformation is to prepare an instantiated template for the following composition (see next section), while taking addition classes into account.

The transformation steps below are applied to a template `T` when `T` is instantiated in a package or another template. `T` is a strongly closed template in C-PT, where any renames specified by the `inst` statement have already been performed. The instantiating template or package is, respectively, an open template or package in C-PT.

The following transformation steps are thus dependent on the instantiations, and concern issues that revolve around addition classes and overrides. We know the semantic bindings of the strongly closed template `T`, and can utilize these in the transformation. We know the syntactical structure of the open template, and utilize this in the transformation.

For each class `C` in `T`:

1. If a variable[1] named `v` is declared in `C`, and a variable named `v` is also declared in an addition class to `C`, then the variable named `v` in `C` should be given a new fresh name, and every usage of `v` in `T` should be updated to the new fresh name.
2. If a method named `m` is declared in `C`, and an addition class to `C` also defines a method named `m` with identical signature, the latter is considered an override of the former in the addition dimension. This can be resolved purely syntactically. Mark the method in `C` with a `@TOverridden` annotation.

Lemma 3. *When the addition-handling transformation has been carried out on a strongly closed template `T` in C-PT, the resulting template `T'` is also a strongly closed template in C-PT with the same meaning, visible names and bindings, with the exception of shadowed variables, which are given fresh names.*

Proof sketch We consider each of the transformation steps:

1. The fresh name will by definition not clash with any other name, and hence not cause any unintentional rebinding. Since every use-site is updated based on the semantic bindings, the renaming is safe.
2. Adding custom annotations to a method has no observable effect besides for reflective programs.

□

14.4.4 The Composing Transformation

This transformation performs the final step of composing the contents of template classes with the instantiating package or template's classes, thus forming ordinary classes.

In the following, we assume that `U` is an open template or package in C-PT that instantiates a set S of other templates and that every template $T_i \in S$ is a strongly closed template in C-PT.

Preconditions for composition The preconditions deal with merging of classes: For every two classes C_j and C_k with identical names (after renaming) from templates T_l and T_m, $l \neq m$, respectively:

- There cannot be a method in C_j that has an identical (i.e. override-compatible) signature in C_k, or in any of C_k's sub- or superclasses.
- There cannot be a field in C_j that has an identical name as a field C_k.
- If C_j has an `extends` clause, then the `extends` clause of C_k must either refer to a class with the same name as that of C_j or to `Object`.

[1] We know that it is an instance variable since we disallow static declarations.

The transformation

1. For every template $T_i \in S$, for every class $C_j \in T_i$:
   (a) If C_j has a method marked with a `@TOverridden` annotation, delete that method. (The reason for not deleting the method instead of putting the annotation there in the first place, is that this would lead to dangling references.)
   (b) If an addition class with the same name as C_j does not exist in U, create an empty addition class with this name in U.
   (c) Copy the body of C_j and append it to the body of the addition class with the same name in U.
2. For every addition class $A_i \in U$, transform the addition class to an ordinary class (i.e., remove "`adds`").
3. For every `inst`-statement $I_i \in U$, remove I_i.

Lemma 4. *When all the transformation steps above are completed for every instantiation, the instantiating template or package is either itself a closed template in C-PT, or a valid (closed) Java package, with the same visible names and bindings, with the exception of variables shadowed by like-named variables and explicit template method overrides in addition classes.*

Proof sketch We consider each of the transformation steps:

1. (a) We know from the previous instantiating transformation that a method that has a `@TOverridden` annotation will have an identical signature in the addition class. It is type-safe to replace a method with another method with an identical signature.
   (b) Creating an empty addition class for a known instantiated template class when none previously exists is trivially safe.
   (c) Appending the body of one class to another is safe since we ensure that there are no name conflicts, and thus no rebinding of variables or method calls.
2. An addition class at this stage in the transformation has no other semantics than an ordinary class, and thus removing the `adds` keyword qualification is trivially safe.
3. At this point in the transformation, the `inst` statement is fully processed, so removing it is safe.

□

For the proof sketch to be complete, we would also have to show that the order in which the instantiations of templates and packages are processed by the transformations does not affect the semantics or externally visible names of the final program. For lack of space we have not shown this, but in general it comes down to asserting that each instantiation is independent. Given the preconditions and transformations

above, this should be hopefully be intuitively understandable, or at least plausible, to the reader.

Presuming such independence, we have shown that the transformation results either in a correct Java package, or in a correct closed template. When the result is a package, the transformation process is complete, and the package can be utilized in any ordinary Java program. When, on the other hand, the transformation results in a template, we reapply the fortifying transformation, and the result is again a strongly closed template in C-PT to which the renaming, addition-handling and composing transformations can be applied for subsequent instantiations.

14.5 Supporting Full PT

Generics In order to support Java-style generics in template classes, there are a few restrictions we would need to impose. First, there would be obvious restrictions on merging: only classes with the exact same number and bounds of type parameters can be merged. Secondly, method overloads where the only discriminator for overload resolution is one or more generic types cannot be allowed to be added by addition classes. This is because Java might choose a non-generic method implementation even for a call-site that explicitly specifies generic parameters [16, sec. 15.12.2.1].

tsuper and tabstract Full PT contains a mechanism for calling the original method for an override made in a template class by using the `tsuper` keyword. In order to type check `tsuper`-calls with an ordinary Java compiler, we first need to make some syntactic transformations, so that these calls appear to be ordinary method calls. One approach is to transform a method call to "`tsuper.m(...)`" into "`tsuper_m(...)`", and doing corresponding transformations on the called methods. If we label methods with annotations we can then utilize a standard Java compiler to see where each `tsuper`-call binds, and transform these methods (marked with a `@TOverriden` annotation in our transformation) to normal Java methods.

The `tabstract` modifier is utilized for method declarations that need to be concretized by an addition class, at the latest in a package. A `tabstract` modifier can be (syntactically) transformed into an annotation, e.g. `@TAbstract`, and a standard Java compiler can then be utilized.

Static declarations With static declarations, renaming can cause rebinding due to precedence. A declaration that was previously referring to e.g. a static field of a class may become inaccessible by a rename. For instance, consider the situation where a method uses the static `System.out` field, and then another field in this class is renamed to `System`. The original access to `System.out` is now invalid, and since Java lacks a global modifier (akin to e.g. C#'s `global::`), there is no straight-forward way to resolve this in general (though this concrete case could be solved by an explicit package reference). One approach is to disallow renaming of fields to names of known

classes, however, for this to be safe we would need to employ a closed-world assumption.

Constructors In [8] we discuss alternatives for supporting constructors in full PT. The current prototype implementation transforms constructors from template classes to ordinary methods that are subsequently called from the package class' real constructor through `tsuper(...)` calls. This has one important drawback, which is that `final` variables cannot be assigned in template class constructors. Other alternatives include constructor combination schemes, but this on the other hand leads to less flexibility with regard to calling different constructors.

14.6 Related Work

Feature-Oriented Programming (FOP) [25] allows the programmer to develop a set of independent *features*. Objects can subsequently be quite freely composed from such features. Unlike for PT merge, the order of composition of FOP features is significant. The paper outlines two implementation strategies based on source-to-source transformation: one translates features to an ordinary inheritance hierarchy, and the other utilizes wrappers and delegation. This is in contrast to the PT approach of creating just a single Java class for each set of merged template classes and additions.

AHEAD [10] is a general mechanism for synthesizing programs by composing incremental software features. The mechanism is realized by the *AHEAD Tool Suite* (ATS) [9]. ATS allows features (e.g. Java/Jakarta files and other artefacts) to be successively refined in separate files, and provides a strategy for recursively composing features from such files. While the AHEAD methodology is more general in the sense that it can compose many different file types utilizing different strategies, PT provides stronger semantic guarantees for source code composition. For instance, unintentional method overrides or changes to overload resolution as exemplified in Section 14.3.3 are not detected by ATS (however, since ATS is open, such support could probably be added). Furthermore, while ATS, in our understanding, is targeted mainly at incremental development and evolution of programs, a main focus of PT is to support adaption and composition of *previously unrelated*, independently developed, modules. For this functionality, name changes and binding preservation play an integral role.

FeatureHouse [2] partly builds on [10], and provides composition/superimposition based on a language-independent feature model called a *feature structure tree* [3]. As for AHEAD, the semantic guarantees for the composed code are not very strong (e.g. the method override and overload resolution problems are present also for FeatureHouse). Such weaker guarantees might be considered a positive thing (instead of a problem) in terms of the increased flexibility gained, but for PT we instead opt for stronger guarantees and more predictable composition results.

Classboxes [12] for Java [11] allow new clients to non-intrusively update existing classes. The scope of change is controlled by a *classbox*, which is explicitly imported. Thus, while PT allows generation and adaption of new classes from template classes,

classboxes allow certain modifications to existing classes to be performed and controlled. A naïve Java implementation based on source-to-source transformation is presented in the paper, utilizing stack introspection for method selection, but this obviously results in a severe slowdown. The authors propose building a specialized VM to deal with this.

The techniques pioneered with *Featherweight Java* (FJ) [18] are utilized by many researchers to precisely define the semantics and prove the soundness (relative to FJ) of extensions to Java. FJ represents a minimal subset of Java features, which makes rigorous proofs tractable even for somewhat complex extensions. While the FJ approach is clearly more precise than the prose-based one we have taken here, it is our opinion that a formalization or correctness argument that is more readily understandable also to those without background in formal methods has its merits as well. The respective authors behind the works discussed below all utilize FJ to prove properties about their mechanisms.

A formalization of FOP for Java, *Feature Featherweight Java* (FFJ), is presented in [1]. Since FJ, and thus also FFJ, is considerably simpler than Java, omitting e.g. overloading, `super` calls and assignments, the main issues presented w.r.t. binding preservation in our work are effectively avoided.

Expanders [28] allow classes and interfaces to be expanded with new methods and fields in a non-intrusive and statically scoped manner. The expansions are explicitly imported into the current scope by client code. Unlike PT, expanders do not support renaming. Like PT, they preserve existing bindings. The described implementation transforms expander code to plain Java code by creating wrapper classes, with one wrapper for each expanded class which holds a reference to the original unexpanded object. By transforming the source code in this way, the problems with unintentional rebinding are more easily managed compared to the PT approach, but it has a runtime penalty. The PT transformations, on the other hand, typically results in code that is as efficient as plain Java. Furthermore, problems with *object schizophrenia* are an issue when utilizing wrappers for translating to Java code.

JavaGI [29] is an extension to Java inspired by Haskell's type classes. Existing types can be refined by explicit, retroactive, interface implementation declarations. *Multi-headed interfaces* provide support for multi-type concepts in a natural way, exemplified by the Observer pattern [15]; a corresponding implementation in PT could be a template with types for each role, an example can be found in [7]. An implementation of JavaGI via a translation to plain Java is described in [30]. As for expanders, an approach based on wrapper classes that preserves bindings is used. Also, a dispatch mechanism based on dictionary lookup is needed.

MorphJ [17] presents a different approach to templating based on static reflective iteration over the structure of generic parameters to "template" classes. A template can be instantiated with existing classes as actual parameters to generate a new class. Compared to PT, MorphJ offers wider flexibility with regard to the structure of the instantiated classes. On the other hand, PT is more flexible in terms of composition and refinement, and allows templates to consist of class hierarchies. It would be an interesting direction for future work to combine the static reflective capabilities of MorhpJ

with the generic mechanisms recently added to PT [5].

14.7 Concluding Remarks

In this paper, we have defined the semantics of a core subset of the package template mechanism through transformations to plain Java source code. One of the main obstacles to be overcome in such transformations is to prevent accidental rebinding of code for classes that are extended with new methods, or where attributes are renamed, while at the same time retaining performance as good as standard Java and not mangling visible names.

Our approach to this problem is to present a set of transformations that "strengthen" the source code in order to be resilient to rebindings, and then present preconditions for subsequent transformations. Through a series of lemmas we sketch a proof showing that PT programs will always be transformed to correct Java programs in source code form with the original bindings intact.

Acknowledgements

This work has been done within the context of the SWAT project (NFR grant 167172/V30). We would like to thank Birger Møller-Pedersen and the anonymous reviewers for valuable feedback on this paper.

Bibliography

[1] Sven Apel, Christian Kästner, and Christian Lengauer. Feature featherweight java: a calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 101–112, New York, NY, USA, 2008. ACM.

[2] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proc. 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Sven Apel and Christian Lengauer. Superimposition: a language-independent approach to software composition. In *Proc. 7th international conference on Software composition*, SC'08, pages 20–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Eyvind W. Axelsen and Stein Krogdahl. Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In *DLS '09: Proc. 5th symp. Dyn. lang.*, pages 15–26, New York, NY, USA, 2009. ACM.

[5] Eyvind W. Axelsen and Stein Krogdahl. Adaptable generic programming with required type specifications and package templates. In *Proc. AOSD '12*, AOSD '12, pages 83–94, New York, NY, USA, 2012. ACM.

[6] Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller-Pedersen. Controlling dynamic module composition through an extensible meta-level API. In *DLS 2010: Proc. 6th symp. Dyn. lang.*, New York, NY, USA, 2010. ACM.

[7] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09*, pages 37–42, New York, NY, USA, 2009. ACM.

[8] Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. Challenges in the design of the package template mechanism. *Transactions on Aspect-Oriented Programming*, 2012. To appear, available now from `http://swat.project.ifi.uio.no/`.

[9] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. pages 3–35. 2006.

[10] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proc. 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[12] Alexandre Bergel, Stephane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proc. JMLC 2003 (Joint modular languages conference), volume 2789 of LNCS*, pages 122–131. Springer-Verlag, 2003.

[13] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[14] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Oracle America, Inc, 2011.

[17] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with morphj. *SIGPLAN Not.*, 43(6):79–89, June 2008.

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[19] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Obj. Tech.*, 8(7):59–85, 2009.

[20] E. Murphy-Hill and A.P. Black. Refactoring tools: Fitness for purpose. *Software, IEEE*, 25(5):38 –44, sept.-oct. 2008.

[21] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proc. 31st Intl. Conf. Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for java. In *Proc. 12th intl. conf. on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[23] W. Opdyke and R. E. Johnson. Refactoring, an aid in designing application frameworks and evolving object-oriented systems. In *Proc. Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPA)*, 1990.

[24] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[25] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. European Conference on Object-Oriented Programming*, pages 419–443, 1997.

[26] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for java. In *Proc. OOPSLA '08*, OOPSLA '08, pages 277–294, New York, NY, USA, 2008. ACM.

[27] Håkon Stordahl. BooPT: Implementasjon av package templates for Boo. Master's thesis, Dept. of Informatics, Uni. Oslo, Gaustadalléen 23B, Postboks 1080 Blindern, 0316 Oslo, 2012.

[28] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proc. 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 37–56, New York, NY, USA, 2006. ACM.

[29] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized Interfaces for Java. In *ECOOP 2007, Proceedings*, LNCS. Springer-Verlag, July 2007.

[30] Stefan Wehr and Peter Thiemann. Javagi in the battlefield: practical experience with generalized interfaces. In *Proc. eighth international conference on Generative programming and component engineering*, GPCE '09, pages 65–74, New York, NY, USA, 2009. ACM.